```
EEEEEEE    DDDD          A       SSSSSS
EE         DDDDD       AA AA     SS  SS
EEEEE      DD   DD    AA   AA    SS
EEEEE      DD   DD    AAAAAAA    SSSSSS
EE         DD   DD    AA   AA        SS
EEEEEEE    DDDDD      AA   AA    SS  SS
EEEEEEE    DDDD       AA   AA    SSSSSS
```

Model I, Model II, Model III, & LDOS 6.x

Editor Assembler Reference Manual

MISOSYS
P. O. Box 4848
Alexandria, Virginia 22303-0848

N O T I C E

L I M I T E D W A R R A N T Y

MISOSYS shall have no liability or responsibility to the purchaser or any other person, company, or entity with respect to any liability, loss, or damage caused or alleged to have been caused by this product, including but not limited to any interruption of service, loss of business and anticipatory profits, or consequential damages resulting from the operation or use of this program.

Should this program recording or recording media prove to be defective in manufacture, labeling, or packaging, MISOSYS will replace the program upon return of the program package to MISOSYS within 90 days of the date of purchase. Except for this replacement policy, the sale or subsequent use of this program material is without warranty or liability.

W A R N I N G

MISOSYS
P. O. Box 4848
Alexandria, Virginia 22303-0848
703-960-2998

<<**********>>
<<**********>>
<<****** MISOSYS EDAS - Editor Assembler IV *****>>
<<****** Copyright 1980, by Roy Soltoff *****>>
<<**********>>
<<**********>>

Table of Contents

Preface

EDAS is an evolutionary product. It has been designed to provide many useful assembler capabilities for the most discriminating programmer while at the same time, its command syntax and ease of use provide for an excellent assembler language development tool for the programmer from beginner through advanced level. Its editing syntax has been implemented to appear identical to that found in the '80s BASIC interpreter so as to provide a high degree of familiarity and minimal training requirements.

Although considerable effort was expended to make the user reference manual as complete as possible, this documentation package in no way is to be considered an instructive guide into the writing of Z-80 source programs. Many reference texts are available that deal with learning and improving your abilities to program in assembly language. If you are learning assembly language, your reference materials should include at least one of the many good texts on the market, an asortment of periodicals, and a good disassembler.

My advice is to peruse the contents of this reference manual to familiarize yourself with its information and the Editor Assembler's capabilities as well as the Utility Applications included on the distribution diskette. If you have any questions concerning this application, feel free to call or write; however, be prepared to give your EDAS registration number. It would also be helpful to make sure your questions are not answered in the manual.

Speaking of registrations, MISOSYS would like to provide you with the best technical support possible. To provide this support, we need to know who our customers are. So please fill out the registration form packaged with the diskette and return it to us promptly - postal card postage is sufficient. The registration number located on the diskette label must be entered onto the registration card and should also be entered in the space provided below. The registration number must be mentioned on all correspondance with us or when telephoning for service, so don't lose it. Thank you.

Roy Soltoff

EDAS Registration _____

Introduction to EDAS Version IV

DISTRIBUTION DISKS

The Model I/III EDAS IV version and each of its utilities, are single programs that work on both the Model I and III under LDOS. It is distributed on a 35 track single density data diskette. The LDOS 6.x EDAS Version IV is distributed on a 40 track double density data diskette. The Model II Version is distributed on an 8 inch diskette.

It is strongly recommended that before using your new Editor Assembler, you should make a BACKUP copy to use in a working environment and retain the EDAS diskette as your MASTER copy. This "master" should be backed up to produce a "working" copy and the "master" archived. The BACKUP utility procedures are found in your DOS Owner's Manual in the section entitled "UTILITY PROGRAMS". After creating a BACKUP copy of the EDAS diskette, store the MASTER diskette in a safe place. Use only your "working" copy for production.

THE EDAS FACILITY

The MISOSYS Editor Assembler is a RAM-resident text editor and RAM resident or direct disk assembler for the Model I, II, and III microcomputer systems, as well as computers running under LDOS 6.x. The Editor Assembler was designed to provide the maximum in user interface and ease of use while providing capabilities powerful enough for the expert Z-80 assembly language programmer.

The text editing features of the Editor Assembler facilitate the manipulation of alphanumeric text files for both assembler source and compiler source languages. The most common use of the editing capability is in the creation and maintenance of assembly language source programs to be assembled by EDAS. Through full support of upper and lower case text entry, the Editor can serve as a line-oriented text processing tool.

The assembler portion of the Editor Assembler facilitates the translation of Z-80 symbolic language source code programs into machine executable code. This object code may then be executed directly from the DOS Ready prompt.

Although EDAS could be used as an entry-level assembler, the scope of the documentation assumes a previous knowledge of assembler language and the hexadecimal number system. This is not a "learning" manual - it details the use of EDAS Version IV but in no way attempts to teach you how to program in the Z-80 assembly language. You should have available a standard reference handbook on the Z-80 code. Many texts are available.

The <A>ssemble command supports the assembler language specifications set forth in the ZILOG "Z80-ASSEMBLY LANGUAGE PROGRAM MANUAL", 3.0 D.S., REL.2.1, FEB 1977, with certain limitations.

Nested MACROs are supported; however, MACROs must be defined individually.

Operand expressions may contain the "+", "-", "*", "/", ".MOD.", "&" or ".AND." (logical AND), "!" or ".OR." (logical OR), ".XOR." (logical XOR), .NOT. (logical ones complement), .NE. and .EQ. (logical comparison, and "<" (shift) operators, and are evaluated on a strictly left to right basis. Parentheses are not allowed!

Conditional assembly commands, where a programmer may control which portions of the source code are assembled, are implemented with the conditional pseudo-ops; IF, IFLT, IFEQ, IFGT, IFNE, IFLT$, IFEQ$, IFGT$, IFNE$, IFDEF, IFNDEF, and IFREF.

Constants may be decimal (D), hexadecimal (H), octal (O) or (Q), binary (B), or string ('cc').

The Assembler commands supported are *LIST OFF, *LIST ON, *MODULE, *PREFIX, *GET filespec, and *SEARCH library, as well as a range of listing pseudo-ops (TITLE, SUBTTL, SPACE, PAGE, and constant declarations for bytes, words, and strings).

A label can contain only alphanumeric characters and certain special characters. A label can be up to 15 characters long. The first character must be alphabetic (A-Z), the dollar sign ($) or the <AT> sign (@). Subsequent characters must be alphanumeric (A-Z, 0-9) or selected special characters - <AT> sign (@), underline (_), question mark (?) or dollar sign ($). For compatibility with MACRO-80, a colon may be inserted immediately following the symbol.

Two utilities are included with the EDAS application. XREF/CMD is used to generate a full cross reference listing of symbol use. TTD/CMD is a tool to convert EDTASM compatible source cassette files to EDAS source disk files.

NOTATION CONVENTIONS

Braces "{}"

Braces enclose optional information. The braces are never input in Editor Assembler commands (Note: braces are used in C language source code).

Ellipses "..."

The ellipses represents repetition of a previous item.

Line number "line"

"line" represents a number arbitrarily assigned to a statement for the purpose of identifying it to the editor functions. "Line" can be any decimal number ranging from <1 - 65529>.

Period "."

A period may be used in place of any line number. It represents a pointer to the current line of source code being assembled, printed, or edited. It is termed the "current line pointer" throughout this documentation.

Top of Text "#" or "t"

The pound sign character,"#", or the letter "t", may be used in place of any line number during a line number reference. It represents the beginning or top of the text buffer.

Bottom of Text "*" or "b"

The asterisk character, "*", or the letter "b", may be used in place of any line number during a line number reference. It represents the bottom of the text buffer.

Line Increment "inc"

This is a number representing an increment between successive line numbers.

LOWER CASE ENTRY
==========

Lower case is supported freely throughout EDAS for text and command entry. All Editor Assembler commands may be entered in lower case as well as upper case to facilitate its use as a general purpose text editor.

Assembler source code can be entered in upper case or lower case. For lower case entry, the Editor must be in the case converted mode (see the <S>witch case command). This mode automatically converts lower case entry to upper case except for text which is between single quotes (enabling lower case string constants) and for all text following a semicolon (permitting lower case comments).

EXECUTING EDAS

EDAS is a directly executable command file. It is accessed in response to the DOS command prompt simply by entering:

```
EDAS (MEM=val,JCL,ABORT,LC,EXT="ext",Pn=val)
EDAS *
```

MEM=val	is used to protect a high memory region just like you can in BASIC.
JCL	is used when running from Job Control Language so that EDAS uses the @KEYIN routine for its keyboard input.
ABORT	if specified, EDAS will automatically abort after an assembly with errors. It will return to DOS Ready.
LC	is used when editing LC source files. It will set tabs to 4, default extension to "CCC", and invoke "lower case permitted".
EXT="ext"	provides a means by which the default source file extension can be altered to "ext".
Pn=val	can be used to pass symbol equates to the assembler from the command line. "n" can range from <1-4> permitting four symbol equates.
*	if specified, will reload EDAS and maintain the text buffer pointers.

Note: "val" can be entered as parm=ddd or parm=X'hhhh'.

There are no parameter abbreviations.

The parameters shown in parentheses are entirely optional. They are used to alter the behavior of EDAS. Parameters enhance the utility of the Editor Assembler by giving it greater flexibility. These options are used as follows:

MEM=val

This parameter is used to protect a high region of memory from use by EDAS. The region would usually be reserved for an in-memory assembly.

If you do not enter a value, EDAS will recover the value stored at HIGH$ (address X'4Ø49' and X'4Ø4A' for the Model I or X'4411' and X'4412' for the Model III) or use the value returned by the DOS (for the Model II or LDOS 6.x) and use it for top of memory, maintaining its MEMTOP pointer to that value. If you do not wish to protect any memory from use by the Editor Assembler, do not use this parameter.

You may protect a memory region similar to that which can be protected from BASIC by entering a non-zero value. Enter an address value in decimal or hexadecimal which is one byte less than the lowest address you want to protect. Your entry must be greater than the start of the text buffer. At no time will the Editor Assembler use memory higher than the entered value. This function is useful if you have placed a high memory driver or utility program that does not maintain HIGH$ and you want to avoid clobbering it. For example:

```
EDAS (MEM=X'DFFF')
```

will restrict EDAS from using any address above X'DFFF'. Your in-memory program can be assembled starting at address X'EØØØ'.

JCL (LDOS use only)

This parameter will cause EDAS to enable the parsing of certain characters. This hinders the ability of commanding EDAS from within the Job Control Language (JCL) of LDOS. If you want to control the assembly process from JCL, use the JCL parameter in the EDAS command line. If you are going to <I>nsert text while in a JCL mode, then you must use the "%Ø1" to simulate a <BREAK> in the JCL file. Don't forget, the "%Ø1" can only be used if you are going to compile the JCL. For example, the following enters EDAS and inserts one line:

```
edas (jcl)
i
This is a test
%Ø1
//stop
```

ABORT

This parameter will cause EDAS to abort and return to DOS upon an assembly or disk error, or one of the following errors: no text in buffer, line number too large, bad parameters, buffer full, no such line, *GET or *SEARCH error, *SEARCH file not a PDS, PDS member error. . It is useful when running from a Job Control Language to inhibit erroneous jobs from continuing.

LC
--

This parameter is used when you are editing LC source files (C language). It will do three things for you. LC changes the source file default extension from "ASM" to "CCC" - "CCC" is used in the LC compiler. It will change the tab stops from every eight columns to every four columns - more reasonable for LC source code. The LC parameter will also invoke the <S>witch case command to "lower case permitted" as LC source code is entered primarily in lower case.

EXT="ext"

This parameter is available for those using the EDAS editor to edit and maintain files other than EDAS assembler source files. For instance, the M-80 assembler uses "MAC" as the standard extension. FORTRAN uses "FOR". You may be using EDAS to create or edit JCL files. Use this parameter to change the default source file extension (that used with the <L>oad and <W>rite commands) to one of your choice. You must enter a full three characters if you use this parameter. For example:

```
EDAS (EXT="MAC")
```

specifies that "MAC" be used as the default extension (make sure the supplied extension is entered in UPPER CASE).

Note that the override of "CCC" if the LC parameter is used takes precedence. If LC is specified, the EXT= parameter is ignored.

Pn=val

This parameter provides the power of entering symbol table equates directly from the EDAS command line. "Pn" is actually four parameters as "n" can range from <1-4>. Thus, you specify the parameter as either P1, P2, P3, or P4. These parameters are EDAS entry symbol table additions. By passing parameter values with these on the EDAS command line, you can alter four symbol table entries. Thus, you can use these to control EQUate options, pass values to symbols, etc. The format usable is:

Pn	sets @@n to TRUE.
Pn=ddd	sets @@n to decimal value ddd.
Pn=X'hhhh'	sets @@n to hexadecimal value hhhh.

The actual labels added to the symbol table as DEFLs are "@@n", where "n" is the same as the "n" of "Pn". This is depicted as follows:

```
+-------------------------------------------------------------------+
|                                                                   |
|     P1 == @@1     P2 == @@2     P3 == @@3     P4 == @@4         |
|                                                                   |
+-------------------------------------------------------------------+
```

The four symbols initially have a value of zero (logical FALSE). You can use these to externally set flags for use in conditional assembly (or whatever else your heart desires). For example, say you have a program that uses two conditional symbols, MOD1 and MOD3. If your program has the statements:

```
MOD1    EQU   @@1
MOD3    EQU   @@3
```

then an EDAS command line of EDAS (P1) will set "@@1" to TRUE, "@@3" was defaulted to FALSE, and thus "MOD1" would be TRUE and "MOD3" would be FALSE since the two conditional symbols you are using are equated to the "@@n" parameters.

You will find this parameter support a great feature when running EDAS from JCL.

EDAS *

The "EDAS *" is used to re-enter EDAS keeping the source program and variables intact. This permits you to recover after a re-boot providing the Editor Assembler region is not disturbed or in case you inadvertantly entered the <B>ranch command without saving your source file. The region occupied by the Editor Assembler is not normally disturbed by a RESET and boot of DOS. Remember to hold the <ENTER> key depressed during the RESET operation if your SYSTEM diskette contains an AUTO function.

EDAS COMMAND MODE

Once "EDAS" is entered, the following message will appear on the video display screen:

```
MISOSYS EDAS-n.n
```

The "n.n" is indicative of the current version number. This display is followed by a right caret ">" prompt. The prompting character is displayed whenever EDAS is ready to accept a command. Detailed information on all commands supported can be found in the chapter entitled, COMMANDS.

SYNTAX

The basic format of an assembly language statement consists of up to four fields of information. These fields, in order, are:

{LABEL} {OPCODE} {OPERAND{S}} {;COMMENT}	
LABEL	is a symbolic name assigned the address value of the first byte of the object instruction.
OPCODE	is the mnemonic of a specific Z-8Ø assembler instruction or pseudo-OPeration code.
OPERANDS	are arguments of the OPCODE.
;COMMENT	is an informative notation that is ignored by the assembler but aids in documenting the source code.
Note: Fields are separated by a tab or spaces.	

As can be noted from the format box, none of the fields are required; however, each line should contain at least one field. This may seem unusual at first, but it is readily explained. If you want the comment field to occupy the entire line, start the line with a semi-colon in the first character position of the line - then, no other field is needed. A symbolic label can exist by itself on a line. There are some Z-8Ø operation codes that have no arguments; thus, an OPCODE could exist by itself on a line (in field 2). You will never have an argument by itself as an argument relates to an OPCODE.

The statement line is considered to be freely formatted. That means that there are no columnar restrictions. Fields are separated by one or more tabs or spaces. If a tab is used, it makes for neater listings. Tabs are also retained as tabs and thus will keep source files smaller than using multiple spaces.

Symbolic Labels

A label is a symbolic name of a line of code. Labels are always optional. A label is a string of characters no greater than 15 characters. The first character must be a letter (A-Z) or one of the special characters, "$" and "@". The "@" as the first character of a label is useful for highlighting certain labels since labels begining with "@" appear at the beginning of an ascendingly sorted list (such as the symbol table listing or cross-reference listing). The dollar sign is supported for easier adaptation of M-8Ø source files. Actually, the "$" sorts out higher than "@"; however,

it is recommended that you reserve use of "$" as the first character of "local" labels. This can be very useful in light of the "-SL" assemble switch

A label may contain, within character positions 2-15, letters (A-Z), decimal digits (Ø-9), or certain special characters: the <AT> sign, "@"; the underline, "_"; the question mark, "?"; or the dollar sign, "$". The dollar sign "$", appearing by itself, is reserved for the value of the reference counter of the current instruction. It cannot be used as a single character symbol.

A symbol appearing by itself in the LABEL field of a line, will be interpreted as being equated to the current value of the program counter. Thus, the following two LABEL examples are completely equivalent:

```
ALLALONE
ALLALONE   EQU $
```

Certain labels are reserved by the assembler for use in referring to registers. Others are reserved for branching conditions (condition codes) and may not be used for labels. (these conditions apply to status flags). The following labels are reserved and may not be used for other purposes:

Reserved Labels
A, B, C, D, E, H, L, I, R,
IX, IY, SP, AF, BC, DE, HL
C, NC, Z, NZ, M, P, PE, PO
AND, EQ, MOD, NE, NOT, OFF, ON, OR, XOR

Examples of labels:

```
ENTRY       @OPEN       BUFFER$      BYTE_POINTER     WHAT?
SELECT_CODE $$CORE      @            CARRIAGE_RETURN  @EXIT
```

Opcodes

The OPCODES for the EDAS Version IV Assembler correspond to those in the Z-8Ø-ASSEMBLY LANGUAGE PROGRAMMING MANUAL, 3.Ø D.S., REL 2.1, FEB 1977.

Operands

Operands are always one or two values separated by commas. Some instructions may have no operands at all.

A value in parentheses "()" specifies indirect addressing when used with registers, or "contents of" otherwise.

Constants are data declarations of fixed value. They are constructed as a sequence of one or more digits and an optional radix specification character. The digits must be valid for the radix used. The following table denotes aceptable constant composition:

Data Type	Radix Char	Digits	Examples
hexadecimal	H	<Ø-9,A-F>	1AH, ØABH, ØFFH
decimal	D	<Ø-9>	1Ø7D, 1Ø7, 15384
octal	O or Q	<Ø-7>	166Q, 166O
binary	B	<Ø-1>	Ø11Ø111ØB

Note: Decimal is assumed if the radix character is omitted

A constant not followed by one of the radix characters is assumed to be decimal. A constant must begin with a decimal digit. Thus "FFH" is not permitted, while "ØFFH" is valid.

Operands may also be constructed as complicated expressions using the mathematical and logical operators. Due to the extent of the documentation, they are described in the section on "Expressions".

Comments

All comments must begin with a semicolon ";". If a source statement line starts with a semicolon in the first character position of the line, the entire line is a comment. If EDAS is in the lower case converted mode, comments will be retained in whatever case they are entered. It is suggested that comments be entered in lower case with punctuation as required. It will make your source code listings much easier to read. All entry of text following a semi-colon is maintained in its entered case.

EXPRESSIONS

A value of an operand may be an expression consisting of multiple terms (labels and data constants) connected with mathematical operators. These expressions are evaluated in strictly LEFT to RIGHT order. No parentheses are allowed. EDAS does not support operator precedence. Most operators are binary, which means that they require two arguments. Both "+" and "-" have unary uses also. The following operators are supported:

OPERATOR	FUNCTION	EXAMPLE
+	Addition	ALPHA + BETA
-	Subtraction	ALPHA - BETA
*	Multiplication	ALPHA * BETA
/	Division	ALPHA / BETA
.MOD.	Modulo Division	ALPHA .MOD. BETA
<	Shift Left or Right	ALPHA < -BETA
.AND. or &	Logical Bitwise AND	ALPHA .AND. BETA
.OR. or !	Logical Bitwise OR	ALPHA .OR. BETA
.XOR.	Logical Exclusive OR	ALPHA .XOR. BETA
.NOT.	Logical 1's Complement	FALSE EQU .NOT. TRUE
.NE.	Logical Binary Not Equal	ALPHA .NE. BETA
.EQ.	Logical Binary Equal	ALPHA .EQ. BETA
%	Length of MACRO	%#LABEL or %%
%&	MACRO label concatenation	#NAME%&L

Addition (+)

The addition operator will add two constants and/or symbolic values. When used as a unary operator, it simply echoes the value.

Examples:

```
001E    CON30   EQU   30
0010    CON16   EQU   +10H
0003    CON3    EQU   3
002E    A2      EQU   CON30+CON16
```

Subtraction (-)

The minus operator will subtract two constants and/or symbolic values. Unary minus produces a 2's complement.

Examples:

```
000E    A2      EQU   CON30-CON16
FFF2    A4      EQU   -A2
```

Multiplication (*)

The multiplication operator will perform an integer multiplication of a 16-bit multiplicand by an 16-bit multiplier. Overflow of the resulting 16-bit value is not flagged as an error.

Examples:

```
01E0    A5      EQU     CON30*CON16
BF20    A6      EQU     60000*3    ;this overflows
```

Division (/)

The division operator will perform an integer division of a 16-bit dividend by an 8-bit divisor.

Examples:

```
0002    A7      EQU     5/2
1B4D    A8      EQU     48928/7
```

Modulo (.MOD.)

The modulo operator calculates the remainder of the above integer division.

Examples:

```
0001    A9    EQU    5.MOD.2

0005    A10   EQU    48928.MOD.7
```

Shift (<)

This operator can be used to shift a value left or right. The form is:

```
VALUE    <    {-}AMOUNT
```

If AMOUNT is positive, VALUE is shifted left. If AMOUNT is negative, VALUE is shifted right. The magnitude of the shift is determined from the numeric value of AMOUNT. A good use of the SHIFT operator is to determine the high order byte value of a 16-bit value.

Examples:

```
0057    HIORD  EQU    5739H<-8

C000    A1     EQU    3C00H<4

03C0    A2     EQU    3C00H<-4

BBFF    A3     EQU    3CBBH<8+255

03C0    A3     EQU    15+3C00H<-4
```

The next higher page address in a program is easily calculated with:

```
CORE   DEFL   $<-8+1<8
       ORG    CORE
```

Logical AND (.AND. or &)

The logical AND operator bitwise ANDS two constants and/or symbolic values. Each bit position of the 16-bit resultant value is a "1" only if both

arguments have a "1" in the corresponding position, or a "Ø" if either argument has a "Ø".

Examples:

```
        3CØØ    A1      EQU     3CØØH&ØFFH

        ØØØØ    A2      EQU     Ø&15

        ØØØØ    A3      EQU     ØAAAAH.AND.5555H
```

Logical OR (.OR. or !)

The logical OR operator bitwise "ORS" two constants and/or symbolic values. Each bit position of the 16-bit resultant value is a "1" if either argument has a "1" in the corresponding position, or a "Ø" if neither argument has a "1".

Examples:

```
        3CFF    A1      EQU     3CØØH!ØFFH

        ØØØF    A2      EQU     Ø.OR.15

        FFFF    A3      EQU     ØAAAAH.OR.5555H
```

Logical XOR (.XOR.)

The logical XOR operator performs a bitwise exclusive OR on two constants and/or symbolic values. Each bit position of the 16-bit resultant value is a "1" only if both arguments have reversed bits in the corresponding position (i.e. one must have a "1" while the other must have a "Ø"). The XOR operation is considered a modulo two addition.

Examples:

```
        3CF8    A1      EQU     3CØ7H.XOR.ØFFH

        ØØØ7    A2      EQU     8.XOR.15

        FFFF    A3      EQU     ØAAAAH.XOR.5555H
```

Logical NOT (.NOT.)

This is a unary operator. It performs a one's complement on the term it precedes. Observe the following examples:

```
FFFE    T1    EQU   .NOT.1
FFFF    T2    EQU   .NOT.Ø
ØØØØ    T3    EQU   .NOT.-1
```

Logical NOT-EQUAL (.NE.)

This operator is a binary operator that compares two adjacent terms. The resultant value is TRUE if the terms are not equal. A FALSE result is returned if the two terms are equal. Observe the following examples:

```
ØØØØ    T1    EQU   1ØØØ.NE.1ØØØ
FFFF    T2    EQU   1ØØØ.NE.1Ø
FFFF    T3    EQU   1.NE.-1
ØØØØ    T4    EQU   .NOT.Ø.NE.-1
```

Logical EQUAL (.EQ.)

This operator is a binary operator that compares two adjacent terms. The resultant value is TRUE if the terms are equal. A FALSE result is returned if the two terms are not equal. Observe the following examples:

```
FFFF    T1    EQU   1ØØØ.EQ.1ØØØ
ØØØØ    T2    EQU   1ØØØ.EQ.1Ø
ØØØØ    T3    EQU   1.EQ.-1
FFFF    T4    EQU   .NOT.Ø.EQ.-1
```

Macro Length Operator (%)

The length operator is applicable only with MACRO usage. Therefore, its use will be discussed in the chapter on MACRO PROCESSING.

Z-8Ø STATUS INDICATORS (FLAGS)

The flag registers (F and F') supply information to the user regarding the status of the Z-8Ø at any given time. The bit positions for each flag are as follows:

7	6	5	4	3	2	1	Ø
S	Z	X	H	X	P/V	N	C

C is the Carry flag.
N is the Add/Subtract flag.
P/V is the Parity/Overflow flag.
H is the Half-carry flag.
Z is the Zero flag.
S is the Sign flag.
X is not used.

Each of the two Z-8Ø flag registers contain six (6) bits of status information which are set or reset by CPU operations. Four of these bits are testable (C, P/V, Z, and S) for use with conditional jump, call, or return instructions. Two flags (H, N) are not directly testable and are used by the Z-8Ø internally to handle Binary Coded Decimal (BCD) arithmetic. Two flag register bits (3, 5) are not used by the Z-8Ø.

In the Z-8Ø mnemonic instruction set, the "CALL", "JP", and "JR" instructions can contain a "condition code" which is part of the argument of the OPCODE. The branching determination is performed according to the result of the flag register testable bits. The mnemonics for these condition codes are as follows:

FLAG	CONDITION SET	CONDITION NOT SET
Carry	C	NC
Zero	Z	NZ
Sign	M (minus)	P (plus)
Parity	PE (even)	PO (odd)

Carry Flag (C)

The carry flag is set or reset depending on the operation being performed. For "ADD" instructions that generate a carry and "SUBTRACT" instructions that generate a borrow, the carry flag will be set. The carry flag is reset by an "ADD" that does not generate a carry and a "SUBTRACT" that generates no borrow. This saved carry facilitates software routines for extended precision arithmetic. Also, the "DAA" instruction will set the carry flag if the conditions for making the decimal adjustment are met.

For instructions RLA, RRA, RLS, and RRS, the carry bit is used as a link between the least significant bit (LSB) and most significant bit (MSB) for any register or memory location. During instructions RLCA, RLC s and SLA s, the carry contains the last value shifted out of Bit 7 of any register or memory location. During instructions RRCA, RRC s, SRA s, and SRL s, the carry contains the last value shifted out of Bit Ø of any register or memory location.

For the logical instructions AND s, OR s, and XOR s, the carry flag will be reset. The carry flag can also be set (SCF) or complemented (CCF).

Add/Subtract Flag (N)

This flag is used by the decimal adjust accumulator instruction (DAA) to distinguish between "ADD" and "SUBTRACT" instructions. For all "ADD" instructions, "N" will be set to a "zero". For all "SUBTRACT" instuctions, "N" will be set to a "one".

Parity/Overflow Flag (P/O)

This flag is set to a particular state depending on the operation being performed. For arithmetic operations, this flag indicates an overflow condition when the Accumulator result is greater than the maximum possible number (+127) or is less than the minimum possible number (-128). The overflow condition is determined by examining the sign bits of the operands.

For addition, operands with different signs will never cause overflow. When adding operands with like signs and the result has a different sign, the overflow flag is set. For example:

```
+12Ø  =  Ø111 1ØØØ     ADDEND
+1Ø5  =  Ø11Ø 1ØØ1     AUGEND
-------------------------------
+225  =  111Ø ØØØ1    (-95) SUM
```

The two numbers added together have resulted in a number that exceeds +127 and the two positive operands have resulted in a negative number (-95) which is incorrect. The overflow flag is therefore set.

For subtraction, overflow can occur for operands of unlike signs. Operands of like sign will never cause overflow. For example:

```
  +127  =  Ø111 1111    MINUEND
(-)-64  =  1ØØ ØØØØ    SUBTRAHEND
-----------------------------------
  +191  =  1Ø11 1111    DIFFERENCE
```

The minuend sign has changed from a positive to a negative giving an incorrect difference. The overflow flag is therefore set. Another method for predicting an overflow is to observe the carry into and out of the sign bit. If there is a carry in and no carry out, or if there is no carry in and a carry out, then overflow has occurred.

This flag is used with logical operations and rotate instructions to indicate the parity of the result. The number of "one" bits in a byte are counted. If the total is odd, "ODD" parity (P=Ø) is flagged. If the total is even, "EVEN" parity is flagged (P=1). When inputting a byte from an I/O device "IN r,(C)", the flag will indicate the parity of the data.

During search instructions (CPI, CPIR, CPD, and CPDR) and block transfer instructions (LDI, LDIR, LDD, and LDDR), the P/V flag monitors the state of the byte count register (BC), When decrementing the byte counter results in a zero value, the flag is reset to zero, otherwise the flag is a one.

During "LD A,I" and "LD A,R" instructions, the P/V flag will be set with the contents of the interrupt enable flip-flop (IFF2) for storage or testing.

The Half Carry Flag (H)

The half carry flag (H) will be set or reset depending on the carry and borrow status between bits 3 and 4 of an 8-bit arithmetic operation. This flag is used by the decimal adjust accumulator instruction (DAA) to correct the result of a packed BCD add or subtract operation. The "H" flag will be set (1) or reset (Ø) according to the following table:

H	ADD	SUBTRACT
1	There is a carry from Bit 3 to Bit 4	There is no borrow from Bit 4
Ø	There is no carry from Bit 3 to Bit 4	There is a borrow from Bit 4

The Zero Flag (Z)

The Zero flag (Z) is set or reset if the result generated by the execution of a certain instruction is a zero. For 8-bit arithmetic and logical operations, the "Z" flag will be set to a "one" if the resulting byte in the Accumulator is zero.

For compare (search) instructions, the "Z" flag will be set to a "one" if a comparison is found between the value in the Accumulator and the memory location pointed to by the contents of the register pair HL.

When testing a bit in a register or memory location, the "Z" flag will contain the state of the indicated bit.

When inputing or outputing a byte between a memory location and an I/O device (INI, IND, OUTI, or OUTD), if the result of register B minus one (1) is zero, the Z flag is set, otherwise it is reset. Also for byte inputs from I/O devices using "IN r,(C)", the Z flag is set to indicate a zero byte input.

The Sign Flag (S)

The Sign flag (S) stores the state of the most significant bit of the accumulator (Bit 7). When the Z-8Ø performs arithmetic operations on signed numbers, binary two's complement notation is used to represent and process numeric information. A positive number is identified by a "zero" in bit 7. A negative number is identified by a "one". The binary equivalent of the magnitude of a positive number is stored in bits Ø to 6 for a total range of from Ø to 127. A negative number is represented by the two's complement of the equivalent positive number. The total range for negative numbers is from -1 to -128.

When inputing a byte from an I/O device to a register, "IN r,(C)", the "S" flag will indicate either positive (S=Ø) or negative (S=1) data.

PSEUDO-OPS

There are many pseudo-OPs which EDAS will recognize. These assembler operations, although written much like processor instructions, interface to the assembler instead of the Z-80 processor. They direct the assembler to perform specific tasks during the assembly process but have no meaning to the Z-80 processor. Some of these pseudo-OPs generate data values used by your program and are called "data declaration" pseudo-OPs. Others control paging operations and may be best termed, "listing" pseudo-OPs. A broad range of operations to invoke the assembly of blocks of code based on conditional evaluations are supported through many "conditional" pseudo-OPs. These assembler pseudo-OPs are:

Constant Declarations	
DB	specifies a data byte or string of bytes. Also equivalent to DEFB, DEFM, and DM.
DC	specifies a multiple of byte constants.
DS	reserves a region of storage for program use. Equivalent to DEFS.
DW	specifies a word (16-bit data value) or a sequence of words. Also equivalent to DEFW.

Origins and Values	
DEFL	establishes a value for a label which can be altered during the assembly.
END	signifies the end of a *GET or *SEARCH member. Will indicate the end of the assembly when detected in the text buffer. Supplies the execution transfer address.
EQU	estalishes a constant value for a label.
LORG	establishes a load origin for executable object code files.
ORG	establishes an execution origin for executable object code files or in-memory assemblies.

Conditionals	
IF	conditional evaluation of expression.
IFEQ{$}	logically TRUE if expression1 = expression2.
IFLT{$}	logically TRUE if expression1 < expression2.
IFGT{$}	logically TRUE if expression1 > expression2.
IFNE{$}	logically TRUE if expression1 <> expression2.
IFDEF	logically TRUE if "label" has been defined prior to this statement, else FALSE.
IFNDEF	logically TRUE if "label" has not been defined prior to the statement, else FALSE.
IFREF	logically TRUE if "label" has been referenced but not defined prior to the statement, else FALSE.
ELSE	alternate clause to be assembled if the prior clause has evaluated TRUE.
ENDIF	signifies the end of a conditional block.
Note: "{$}" denotes alternate macro string comparison.	

Miscellaneous	
COM	generates an object code file comment record.
ENDM	designates the end of a MACRO model.
ERR	forces an assembly error.
MACRO	desigantes the prototype of a MACRO model.
PAGE	transmits a form feed during a listing.
SPACE	generates extra line feeds during a listing.
SUBTTL	invokes a heading sub-title for listings.
TITLE	invokes a heading title for listings.

PSEUDO-OP DB

The "DB" pseudo-OP is used to define a data byte or series of bytes. Its syntax is:

DB	n{,n}{,'c'}{,s}{,expression}
n	defines the contents of a byte at the current reference counter to be "n".
'c'	defines the content of one byte of memory to be the ASCII representation of character "c".
's'	defines the contents of n bytes of memory to be the ASCII representation of string "s", where "n" is the length of "s" and must be in the range 1-63.
expression	is a mathematical expression which evaluates to a number in the range <Ø-255>.

The constant declaration "DB" permits the concatenation of its data arguments using the comma "," as an argument separator. Data values are denoted according to the specifications in the chapter on ASSEMBLY LANGUAGE INFORMATION.

In order to provide compatibility with constant declarations of other assemblers, EDAS provides other data declarations that are completely equivalent to "DB". The following pseudo-OPs can be used in lieu of "DB": DM, DEFB, DEFM. Because DB, DEFB, DM, and DEFM are exact equivalents and all four are supplied only for ease of transition from other assemblers, each must be contained in the OP-code table used by EDAS. However, only "DB" was selected to be high up in the OP-code table. Since the OP-code table is searched sequentially, the use of "DB" in your source code will produce a slightly faster assembly than use of DEFB, DEFM, or DM.

"DB" string arguments permit two connected single-quotes to indicate a single-quote value PROVIDED that two or more characters precede the 2-quote appearance in the string. For example:

```
DB    'AB''C'
```

will produce the character string: 41 42 27 43. This may have been coded as a complex declaration such as, "'AB',27H,'C'", but the extensive declaration support in EDAS provides the easier specification.

The following partial assembler listing demonstrates the versatility of the expanded constant declarations.

```
0000 54    00070    DB    'This',' ','is',' ','a',' ','test'
     68 69 73 20 69 73 20 61
     20 74 65 73 74
000E 01    00080    DB    1,2,'buckle your shoe',3,4,'close the door'
     02 62 75 63 6B 6C 65 20
     79 6F 75 72 20 73 68 6F
     65 03 04 63 6C 6F 73 65
     20 74 68 65 20 64 6F 6F
     72
0030 54    00090    DB    'This is a tes','t'!80H
     68 69 73 20 69 73 20 61
     20 74 65 73 F4
```

In the last example, note the expression argument specified as,

```
't'!80H
```

Much more complicated expressions could be utilized.

The expansions of the constant (the rows of eight bytes per row) will appear in listings. The expansions may be suppressed from your listings by using the assembler switch, -NE.

PSEUDO-OP DC

This pseudo-OP defines a repetitive constant. Its syntax is:

DC quantity,value	
quantity	specifies how many times that "value" is to be repeated as a data byte. It can be defined as any other data definition: n, expression, 'c'.
value	is the constant to be repeated. As in a "DB" data declaration, the value can be specified as a character, 'c', a numeric value, n, or an expression evaluated to a number in the range <Ø-255>.

The pseudo-OP, "DC", will define a repetitive constant and eliminate the necessity of defining a series of identical data values by long DB specifications. For example, the following two statements are equivalent:

```
DB Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø

DC 16,Ø
```

The latter is much shorter, easier to enter as text, more readable, and takes up less space in its source form.

The "quantity" must range from 1 to 65535 (a zero value will result in 65536). The "value" must be less than 256. With this pseudo-OP, you can generate repetitions of a single constant. For example, say you want to set 1ØØ storage locations to a zero value during the assembly. Insert the statement,

```
DC 1ØØ,Ø
```

and it will be done. A character constant can also be used for "value" as illustrated in the following example:

```
DC 256,'A'
```

which will set the next 256 storage locations to the letter, "A".

The expansions of the constant will appear in listings just as they do in the DB expansion. The expansions may be suppressed from your listings by using the assembler switch, -NE.

PSEUDO-OP DS

This pseudo-OP is used to reserve a quantity of storage locations for use by your program. Its syntax is:

DS nn	
nn	reserves "nn" bytes of memory starting at the current value of the reference counter.

The DS pseudo-OP can also be entered as "DEFS" in order to provide a compatibility with other assemblers that use only "DEFS" to reserve storage locations. For reasons of efficiency as discussed earlier, use of the "DS" in lieu of the "DEFS" will result in slightly faster assemblies. Therefore, it is suggested that if you are transfering over to EDAS from another assembler, globally change all "DEFS" pseudo-OPs to "DS".

The quantity, "nn", can be a data value or an expression. Note that "DS" does not define data values. The "DS" pseudo-OP adds the quantity of storage locations reserved to the current program counter (PC) to calculate a new PC value. When generating an object code file, this action will cause the next assembled byte to create a new load record. The following examples depict various "DS" declarations.

Examples of the DS pseudo-OP

```
FCB   DS   32
```

will define a 32-byte region for later use as a File Control Block. Its origin can then be referenced as "FCB".

```
TABLE DS   TABLE_LENGTH * TABLE_WIDTH
```

will reserve a quantity of storage locations equal to the result of multiplying the two terms, TABLE_LENGTH and TABLE_WIDTH.

If your source code is being assembled with the "-CI" switch, EDAS automatically converts all "DS" declarations into equivalent "DC" declarations using a value equal to zero. The above two examples would therefore be translated to the following:

```
FCB   DC   32,Ø
TABLE DC   TABLE_LENGTH * TABLE_WIDTH,Ø
```

PSEUDO-OP DW

This declaration specifies a 16-bit data value. Its syntax is:

DW nn{,'cc'}{,nn}	
nn	defines the contents of a 2-byte word to be the value, "nn".
'cc'	defines the contents of a 2-byte word to be the characters, 'cc'

The DW pseudo-OP can also be entered as "DEFW" in order to provide a compatibility with other assemblers that use only "DEFW" to declare data words. For reasons of efficiency as discussed earlier, use of the "DW" in lieu of the "DEFW" will result in slightly faster assemblies. Therefore, it is suggested that if you are transfering over to EDAS from another assembler, globally change all "DEFW" pseudo-OPs to "DW".

In the expansion of the data word, its least significant byte is located at the current program reference counter while the most significant byte is located at the reference counter plus one. The data word can be a numeric constant, an expression that evaluates to a 16-bit value, or a character constant of one or two characters. The following examples illustrate various forms of "DW" data declarations.

```
0000 1027  00100    DW    10000,1000,100,10,1
     E803 6400 0A00 0100

000A 6261  00110    DW    'ab'

000C 5200  00120    DW    'R','o','y'
     6F00 7900
```

Note that if a single character is defined as a character constant word, the low-order byte of the word will contain the character value and the high-order byte of the word will be set to zero.

PSEUDO-OP DEFL

The "DEFL" pseudo-OP assigns a value to a label. The value is permitted to be changed during the assembly. The "DEFL" syntax is:

```
label  DEFL  nn
label  DEFL  expression

nn          sets the value of "labeT" to the quantity "nn"

expression  sets the value of "label" to the evaluated
            result of "expression".
```

This declaration is similar to the "EQU" declaration except that the label value is permitted to change during the course of the assembly without producing phase errors (which are generally observed as numerous MULTIPLY DEFINED SYMBOL errors). If the value of "label" is declared by a "DEFL", the declaration can be repeated in the program with different values for the same label. One useful purpose to support this method of coding would be to similate the maintenance of two program reference counters. Observe the following sequence of code:

```
          ... some code
PROG$     DEFL    $             ; Save current program counter
          ORG     DATA$         ; Set PC to data counter
MSG1      DB      'This is a test message',CR
DATA$     DEFL    $             ; Save current data counter
          ORG     PROG$         ; Reset PC to program counter
          ... more code
PROG$     DEFL    $             ; Save current program counter
          ORG     DATA$         ; Now set PC to the data counter
MSG2      DB      'Another message',LF,CR
DATA$     DEFL    $             ; Save new current data counter
          ORG     PROG$         ;  then re-establish PC
          ... continuation of program code
```

The program maintains two address counters. One is utilized as a counter to keep track of the code portion of the program (PROG$), while the other is used to keep track of the data portion of the program (DATA$). This technique can be used to keep the data fields associated with routines in close proximity to their associated routine in the source code, while the object code location of the data is collected into some other region.

Labels defined as "DEFL" will be carried as "DEFL" in the EQUate file generation of the Cross-Reference utility. They will also be notated in the cross-reference listing by a plus sign, "+", prefix to the label name.

PSEUDO-OP END

The "END" pseudo is used to denote the exit of a *GET or *SEARCH process, or when used in the memory text buffer, it will denote the end of the source code. Its syntax is:

```
END {nn}
END {label}

        signifies the end of the source program (see
        text for handling during *GET and *SEARCH).

nn      specifies an execution transfer address branch
        that will be used by the system loader.

label   specifies an execution transfer address branch
        to be the value of "label".
```

The "END" statement is used to indicate to the assembler, when the last source code statement is reached so that any following statements are ignored. If no "END" statement is found, a warning is produced. The END statement can specify a transfer address (i.e. END LABEL or END 6ØØØH). The transfer address is used by the DOS program execution to transfer control to the address specified in the END statement. Note that the END statement cannot have a label in the label field of the statement).

The "END" statement is treated differently if detected while assembling a file that was the target of a "*GET filespec" or "*SEARCH library". In the case of the *GET, the "END" is treated as if the end-of-file was reached and EDAS will switch back to assemble from what ever invoked the *GET. A similar process takes place with the *SEARCH, except that EDAS continues the searching process in its normal manner.

PSEUDO-OP EQU

This pseudo-OP assigns a constant value to a label. Its syntax is:

```
label  EQU  nn
label  EQU  expression

nn          sets the value of label to nn.

expression  sets the value of label to the calculated
            value of "expression"
```

The "EQU" (equate) pseudo-OP is the generally accepted way to define constant values for use in your program. This declaration serves a different purpose than the the data declarations such as DB, DC, and DW. Data declarations specify storage locations that contain the values declared. The "EQU" assigns the value to the label; thus, anywhere the label is used, the assigned value is utilized. Your programs will be more readable, and easier to maintain if the values need to be altered in a program revision. For instance, the first starting address of a video memory area might be X'3CØØ' or 15360. If your program had a routine to blank out this video area, it could be written as <A>:

```
CLEAR  LD    HL,15360                    CLEAR  LD    HL,SCREEN
       LD    DE,15360+1                         LD    DE,SCREEN+1
       LD    (HL),' '     <A>  or  <B>          LD    HL,' '
       LD    BC,1Ø23                            LD    BC,CRTLEN-1
       LDIR                                     LDIR
       RET                                      RET
```

If you had established labels for the video screen with: "SCREEN EQU 15360" and "CRTLEN EQU 1Ø24", then the above routine could be re-written as in <B> which not only makes it more readable, but when you revise your program for one that has video memory at a different address, all you need do is change the value of one "EQU" statement.

It is also useful to establish a series of equates for system vectors that are to be used in your program. Don't code a statement as "CALL 4424H"; establish a label such as "@OPEN EQU 4424H", then your CALL statement is coded as "CALL @OPEN", certainly much more readable.

An "EQU" can occur only once for any label. A multiple "EQU" with different values will result in the MULTIPLY DEFINED SYMBOL error.

PSEUDO-OP LORG

The "LORG" pseudo-OP is used to establish an object code file (or part of one) that loads at an address different from where it will execute. The syntax of "LORG" is:

```
        LORG  nn
        LORG  expression

  nn            is the address to start loading the object
                file (or part of the file).

  expression    when evaluated, "expression" will be treated
                the same as "nn".
```

A load-origin assembler directive, "LORG", is provided to cause the load addresses of the object file to be based on the LORG operand while the execution code address references will still be based on the "ORG" operand. This is useful to construct a module (or part of a module) that will load at an address different from its execution address. For example:

```
        ORG   5200H
        LORG  7000H
```

will assemble code so that absolute address references and the execution addresses are referenced from X'5200'; however, the object code file will start loading at X'7000'. Any subsequent "ORG" will maintain the offset difference established at the previous "ORG" until another "LORG" is detected.

Why incorporate such a facility into the assembler? How can I make use of it in my programs? Easy answer! Consider this scenario. A program is composed of three large modules, A, B, and C. Module "A" performs initialization, has "run-time" routines, and determines whether module "B" or "C" is to be executed. Consider further, that once either module "B" or "C" execute, the program terminates. If we assemble all three modules so that they are contiguous to each other, their execution take up more space than is actually needed. If we need to maximize the amount of memory available for data storage, buffers, and stack, we could use an "LORG" to have module "C" load after module "B", but "ORG" module "C" so that it executes where module "B" executes. When module "A" determines that it needs to execute module "C", it can move the entire module in memory to "B's" position easily with an LDIR instruction. This will free up memory which can be used for the needed storage.

PSEUDO-OP ORG

The "ORG" pseudo-OP is used to establish an address for the program counter so that the absolute address references within a program are designated. The syntax of "ORG" is:

```
        ORG  nn
        ORG  expression

nn          sets the address reference counter to the
            value "nn".

expression  when evaluated, "expression" will be treated
            the same as "nn". Terms of "expression" must
            be defined prior to the "ORG" statement.
```

The "ORG" statement is used to tell the assembler at what address to begin generating the object code for statements which follow. The assembler will generate object code starting at the address specified by "nn" or "expression", automatically advancing the program counter by the length of each instruction or data declaration assembled. The "DS" data declaration advances the program counter by the amount of storage locations reserved.

A program can have more than one "ORG" statement. If multiple "ORGs" are used, and one or more inadvertantly will cause the overwrite of a previously assembled module of code, no warning message of any kind will be issued. It is left up to the programmer, to protect against such events by use of conditional tests (using conditional pseudo-OPs) and the "ERR" pseudo-OP.

The ORG pseudo-OP causes no code generation itself but just prepares the assembly process to start a new object deck record with the generation of subsequent object code (note that if the evaluated address is one greater than the current PC, a new object deck record will not be started).

CONDITIONAL PSEUDO-OPS

The "conditional" pseudo-OPs provide a powerful way to maintain a program that is slightly different when assembled to execute on different machine configurations. Instead of having to maintain multiple copies of a program, with each copy having some routines and modifications to make a "custom" version of the program, by using the conditional pseudo-OPs, you can maintain one set of source code that has conditional segments (or blocks) of code that perform the "customization". It is very easy to specify which segments are to be assembled during a particular assembly. The structure of a conditional block is as follows:

```
        IFxx  argument of IF
        .
        code block or segment
        .
        ENDIF
```

The argument of the "IF" takes on different formats depending on the particular "IF" pseudo-OP. It can be an expression, a label, or two expressions separated by commas. More on this later; for now, just refer to it as the argument. If the argument is evaluated to a non-zero value, it is interpreted as a logical TRUE condition. If the argument is evaluated to a zero value, it is interpreted as a logical FALSE condition. When the condition is TRUE, the conditional segment between the "IF" and the "ENDIF" is assembled. If "expression" is evaluated to a zero value then the conditional block is not assembled but just listed (during the listing pass). For the sake of uniformity, use the value of "-1" for a logical TRUE and a "Ø" for a logical false so that, "FALSE EQU .NOT.TRUE" is a valid statment. These can be set as equates in the beginning of a program as follows:

```
TRUE    EQU    -1
FALSE   EQU     Ø
MOD1    EQU     TRUE
MOD2    EQU     FALSE
MOD3    EQU     FALSE
```

BE CAUTIOUS WHERE THE OPERANDS OF THE CONDITIONAL ARE NOT DEFINED PRIOR TO THE "IF". THE CONDITIONAL BLOCK WILL MOST LIKELY EVALUATE "FALSE" ON PASS 1 AND "TRUE" ON PASS 2 OR 3.

Consider a program designed for execution on the Model's I, II, or III computer with different versions for each. The code blocks particular to a Model may be included in one set of source files but established as conditional blocks. For example:

```
IF    MOD1!MOD3
block of code for Model I or Model III
ENDIF
IF    MOD2
block of code for Model II
ENDIF
```

and all that is neccessary to invoke a "custom" assembly is to set one of the conditional "switches" to TRUE and the others, FALSE.

Conditional segments can also be nested, in case complicated logical constructs are needed or in case a conditional segment itself has a conditional sub-segment. For example:

```
IF    expression1
      IF   expression2
      ENDIF
ENDIF
```

is a two-level conditional. Conditional segments can be nested to sixteen (16) levels although you will rarely find a need for more than three.

The conditional construct of IF-ELSE-ENDIF may be used. It is coded as follows:

```
IF    expression
code block 1.
ELSE
code block 2.
ENDIF
```

which implies that if "expression" is TRUE, code block 1 assembles. If "expression" is FALSE, then code block 2 will be assembled. The ELSE construct is not required in a conditional but may be used where you have two alternative segments that can be based on one switch. For instance, if your program has only two "switches", GO and NOGO, your constructs could be either of the following:

```
IF    GO
code block 1
ENDIF
IF    NOGO
code block 2
ENDIF
```

```
IF    GO
code block 1
ELSE
code block 2
ENDIF
```

As mentioned earlier, the IF argument can take one of three forms. The conditional structures of these are as follows:

```
---Type I---      -----Type II------      --Type III--
IF    exp         IFxx{$}  exp1,exp2      IFyy   label
.                 .                       .
code segment      code segment            code segment
.                 .                       .
ENDIF             ENDIF                   ENDIF
```

"xx" can be "LT", "EQ", or "GT" representing less than, equal to, or greater than conditions respectively when comparing "exp1" to "exp2".

{$} The "$" is specified in macro comparisons with the expressions treated as strings (see the chapter on MACRO PROCCESSING).

"yy" can be "DEF", "NDEF", or "REF" representing whether "label" has been defined, undefined, or referenced but undefined.

Type II - IFxx

The Type I constructs have already been explained in detail. Among the Type II constructs, using "IFLT", if the value of expression 1 is less than the value of expression 2, then the conditional code segment will be assembled. Using "IFEQ", the conditional code segment will be assembled only if expression 1 and expression 2 have equal values. The "IFGT" pseudo-OP will assemble the conditional code segment (i.e. result in a TRUE condition) only if expression 1 has a value exceeding that of expression 2. The last possibility is "IFNE", which will cause the assembly of the conditional segment if the expressions are not of equal value.

If, for instance, you want to ensure that a program does not assemble code past a particular address (maybe it would clobber another routine), then the ERR pseudo-op could be used in conjunction with IFGT to force an assembly error as follows:

```
IFGT  $,MAXADDRESS
ERR    Program is too long!
ENDIF
```

which compares the current value of the program counter (PC) to some previously specified maximum address. Once the PC exceeds this maximum value, the condition evaluates TRUE resulting in an assembly of the segment. The "ERR" pseudo-OP is used to force an assembly error.

Type III - IFyy

Among the Type III constructs, "IFDEF LABEL" will evaluate TRUE if "LABEL" has been defined prior to the evaluation of the IFDEF on each assembler pass. "IFNDEF LABEL" will evaluate TRUE if "LABEL" has NOT been defined prior to the evaluation of the IFNDEF on each assembler pass. "IFREF LABEL" will evaluate TRUE if "LABEL" has been referenced but NOT defined prior to the evaluation of the IFREF on each assembler pass.

The Type III constructs will find greater use when working with libraries of code. For instance, if a code segment is a specific routine and is surrounded with an IFREF-ENDIF conditional, the routine will only be assembled if prior to the segment, the "label" has been referenced but not yet defined. If "label" is the entry point symbol to the routine, then the routine will be assembled if it is needed. In a similar manner, you may have a library routine that is always to be placed in your program unless its "label" has already been defined in some alternate routine. Surrounding it with the IFDEF-ENDIF conditional will inhibit its assembly if your program has defined that label.

Suppressing FALSE Conditionals

If during the listing pass, you want to suppress the listing of certain conditional segments that are not assembled (i.e. they are evaluated as FALSE), use the following sequence of operators:

```
*LIST OFF
IF    expression
*LIST ON
code segment
*LIST OFF
ENDIF
*LIST ON
```

With this sequence, the "IF" and "ENDIF" lines will always be suppressed. The conditional block will only be listed if the condition being evaluated is logically TRUE. If all FALSE conditional segments are not to be listed, then you may use the assembler "-NC" switch which inhibits the listing of all FALSE conditionals - including the IF-ENDIF statements.

ENDIF

Very little has been said about the "ENDIF" statement. Very little need be said. Each "IF" statement must be matched up with a corresponding "ENDIF". The "ENDIF" is needed to define the scope of the conditional code block.

PSEUDO-OP COM

This pseudo-OP is used to generate a comment record in the object code file. Its syntax is:

```
COM   <string>

<string>   is the information to be placed as a comment.
```

An object deck comment block can be generated within the executable object code file directly by using the "COM" pseudo-OP. The comment string must have a length less than 128 characters. As can be noted, the comment string must be enclosed in angle brackets. The closing bracket may be omitted. If lower case characters are desired, then single quotes must surround the angle brackets. Neither the quotes nor the angle brackets will be a part of the comment record.

The "COM" pseudo-OP will generate a comment block in the object file of the format X'1F' followed by the string length, followed by the string itself. A typical use would be to place a non-loading copyright statement in an executable object code file. For example:

```
COM   '<Copyright (c) 1982 by Roy Soltoff>'
```

will produce the comment record which would be viewed if the file were listed.

The generation of the "COM" object code record will be inhibited if the assembly is performed using the "-CI" switch. A binary core-image file can not have a non-loadable record.

PSEUDO-OP ERR

The "ERR" pseudo-OP is used to force an assembly error. Its syntax is:

```
ERR {message}

message    is an optional message to inform what is wrong.
```

This pseudo-OP forces an immediate warning error and displays the optional message. It is commonly used in a conditional block for error trapping.

PSEUDO-OP MACRO

The MACRO pseudo-OP is used to define the prototype of a MACRO model. Its syntax is:

```
mname  MACRO  {#parm1}{=dflt1}{,#parm2{=dflt2}}{,...}

mname       is the MACRO name used to refer to the MACRO

#parmn      are dummy parameters of the MACRO which will be
            replaced by actual parameters during the MACRO
            invocation.

dfltn       are optional defaults to be used for the dummy
            parameters when a parameter is not provided in
            the MACRO invocation.
```

MACROs are an extremely powerful tool in an assembler. It provides great convenience in writing programs in building block form. For this reason, an entire chapter has been devoted to MACROs. You should refer to the chapter entitled, MACRO PROCESSING, for information concerning the use of MACROs. Suffice it to say here that MACRO invocations can be nested to eight levels, parameters may be passed by position or by keyword, and a special operator is available to test the length of parameter substitutions.

PSEUDO-OP ENDM

This pseudo-OP is used to specify the scope of a MACRO model. It is used much like the "ENDIF". Its syntax is:

```
mname  MACRO  parms
       model statements
       ENDM
```

LISTING PSEUDO-OPS

Four pseudo-OPs are available to control the assembler listings. These are: PAGE, SPACE, SUBTTL, and TITLE. Their syntax is:

PAGE	{OFF}
SPACE	n
SUBTTL	{<string>}
TITLE	<string>
OFF	is an optional parameter for PAGE to suppress the listing of the PAGE statement.
n	specifies how many line feeds to generate.
<string>	is the title or sub-title string to appear in the listing headings.

A new page can be forced to provide separation of routines, modules, etc. by using the "PAGE" pseudo-op. This pseudo-OP will be ignored if it appears between *LIST OFF and *LIST ON. "PAGE" accepts an operand of "OFF" to suppress the listing of the line containing the PAGE pseudo-OP (i.e. "PAGE OFF" will issue the form feed but suppress printing of the line containing the "PAGE" pseudo-OP).

"SPACE n" performs line spacing whenever the "SPACE" pseudo-OP is used. When assembled, "n" is the number of lines to space and is interpreted as modulo 256. The line containing the SPACE pseudo-op is not displayed. This pseudo-op also will be ignored if it appears between *LIST OFF and *LIST ON.

A sub-title to a heading is permitted with the "SUBTTL" pseudo-OP. The subtitle string length can be from zero (Ø) to 8Ø characters in length. A zero length indicates that sub-titling is disengaged.

Lower case strings can be maintained by the use of single quotes surrounding the angle brackets. You may change the subtitle by using additional "SUBTTL" pseudo-OPs throughout the text. Subtitles will appear on the first page following the "SUBTTL" pseudo-op. A "PAGE" pseudo-OP following a "SUBTTL" will force the subtitle to appear immediately. If the "SUBTTL" text string is null (of zero length), then subtitling will cease on the subsequent page. A line will also be skipped between the subtitle and first printed text line on the page. Where many *GETs are being used, you may want to establish a sub-title for each to provide a visual indication on the listing. For example:

```
        SUBTTL '<Module B - initialization routines>'
        PAGE   OFF
*GET    MODULEB:1
        SUBTTL '<Module C - data extraction routines>'
        PAGE   OFF
*GET    MODULEC:1
```

will print the sub-title on each page of the listing associated with MODULEB. Ideally, each module should be preceded with a SUBTTL statement.

The "TITLE" pseudo-OP automatically invokes a page heading and adds the title to the headings of assembler listings. The title string is limited to 28 characters and only one "TITLE" is accepted. The left and right carets (angle brackets) must be entered but are not output in the listing - they serve only to delimit your title string. The title line will include the EDAS version, the date and time retrieved from the system, your title string, and a page number [page number is limited to the range <1-255> and will wrap around to zero if more than 255 pages are printed]. For this reason, if you use a title, it is advisable to set DATE and TIME prior to executing the Editor Assembler. A line will be skipped between the title and start of printed text (or subtitle if used). Lower case titles will be maintained by surrounding the angle brackets with single quotes as in:

```
        TITLE   '<This is an UC/lc title>'
```

The first "TITLE" pseudo-OP found in the text will be used for titling. Any other "TITLE" pseudo-ops will be ignored.

ASSEMBLER DIRECTIVES

The MISOSYS Editor Assembler, EDAS Version IV, supports five assembler commands. In contrast to source statements which are translated to machine language, these directives are "conversation" to the assembler. Each directs the assembler to behave in a particular manner or perform a specific function. The directives, by themselves, do not generate any machine language code - they merely act as "commands" to the assembler. Each "command" must start in column one of a source statement line, and must start with an asterisk (*). Only the first character of each directive is significant. The entire directive "word" may be entered, or the directive may be abbreviated to its first character. The assembler directives are:

*GET file	Causes the assembler to begin reading source code from the "file".
*LIST OFF	Causes the assembler listing to be suspended, starting with the next line.
*LIST ON	Causes assembler listing to resume, starting with this line.
*MOD exp	Advances the "module" character substitution string and optionally sets/resets the prefix.
*PREFIX exp	Establishes or disengages a prefix character for the MACRO substitution string.
*SEARCH lib	Invokes an automatic search of the Partitioned Data Set (PDS) "lib" to resolve any undefined references capable of being resolved by PDS assembler source member modules.

*GET filespec

This directive invokes assembly from a source disk file. Its syntax is:

*GET filespec
filespec Causes the assembler to begin reading source code from the file, "filespec".

This directive tells the assembler to temporarily switch its source assembly to the file identified as "filespec", and use it to continue the assembly. A default file extension of "ASM" will be used if none is provided in the directive statement. The file itself can be headered and/or numbered, as EDAS will automatically detect its type and adjust accordingly. When the end-of-file is reached, or an assembly language "END" statement is read, assembly automatically resumes from the next statement following the statement which invoked the "*GET". Any "END" statement read during the *GET process will be ignored as the program end. The only "END" accepted will be that in the text buffer.

"*GETs" can be nested to five (5) levels. That is, a statement in memory can GET a file which GETs a file which GETs a file which GETs a file which GETs a file. This assembler directive is extremely powerful. It can be used to provide the capability of assembling large programs which are stored on disk in modules, since more than one *GET may be in the text buffer or "gotten" file.

The text buffer can be composed of nothing but *GET statements (and one END statement) which will provide maximum space in the text buffer for generation of the symbol table. For example, the following could represent the source linkage needed to assemble a program called "PARMDIR/CMD":

```
; PARMDIR/ASM - 04/07/82
;*=*=*
;     Linkage to assemble PARMDIR
;*=*=*
*GET  PARMDIR1
*GET  PARMDIR2
*GET  PARMDIR3
      END  PARMDIR
```

LIST ON/OFF

This directive is used to suppress the listing of blocks of code. Its syntax is:

*LIST off/on	
OFF	Causes the assembler listing to be suspended, starting with the next statement.
ON	Causes assembler listing to resume, starting with this statement.

The pair of directives, "*LIST OFF" and "LIST ON", can be used to suppress the listing of a block of code. Once the "*LIST OFF" is invoked, all statements following will not be listed to the display or the line printer (if assembler switch -LP is specified). The directive "*LIST ON" re-establishes standard listing. An exception to the suppression is that any assembler source statement containing an assembly error will be listed along with its appropriate error message. In this manner, you can use an "*LIST OFF" directive at the beginning of your assembly source (to suppress all listing) and lines containing errors will be forced to be displayed by EDAS.

Examples of the *LIST directive:

```
*LIST OFF
      DB  'This line will not be displayed!'
*LIST ON

*LIST OFF
      DB  'Only the next line will be displayed!'
      LD  (M,100
*LIST ON
```

*MOD expression

This directive is used to increment a character substitution string for the purpose of simulating local labels. Its syntax is:

*MOD (expression)	
	Advances the "module" character substitution string.
expression	is an optional expression to specify a prefix character to the substitution string or reset the current prefix.

The "*MOD" directive will increment a string replacement variable each time the directive is executed. The string will replace the question mark, "?", character in labels and label references found in any line assembled from a *GET or *SEARCH file. Its use is essentially applicable to subroutine libraries where duplication of labels could occur. By specifying the "*MOD" directive as the first statement of each module of code and by using a question mark in labels, you can construct source subroutine libraries for use in your programs without having to worry about duplicate labels occuring. Unless at least one "*MOD" statement is specified, the question mark will not be translated.

Labels such as $?ØØ1 will have the "?" replaced with the current "MOD" string value. Thus, a "*MOD" directive preceding each module will force $?ØØ1 labels in each module to be distinctly named by having the question mark replaced with the substitution string. The "MOD" string value cycles from A-Z, then from AA-AZ, BA-BZ, ..., ZA-ZZ. This will allow for a simulation of "local" labels. Remember, the "?" substitutions will only be made to those source lines fetched from a *GET or *SEARCH file, not from statements resident in memory! It really was designed that way folk's, it's not just a limitation.

If you need more than the 7Ø2 unique string values generated by a single/dual alphabetic string (26*26+26), you will have to specify a "MOD prefix". The prefix invokes a user-specified third character for the substitution string. The "*MOD" directive provides for the assignment of the character prefix to the substitution string. You control the prefix. For example:

```
*MOD    '$'
```

assigns the character "$" to prefix all "MOD" substitutions. Once invoked, you can change to any other character by another "*MOD" command or remove the prefix by entering an expression whose value is zero.

*PREFIX expression

This directive gives you the capability of specifying a constant third character to the MACRO substitution string. Its syntax is:

*PREFIX expression
expression establishes or disengages a prefix character for the MACRO substitution string.

The Macro substitution string can be prefixed with a user-entered character constant. This is achieved by using the "*PREFIX" assembler directive. The expression character or value entered in field two becomes the prefix character. It must be a character that is valid for assembler source labels. For example,

```
*PREFIX '$'
```

will cause MACRO local label string substitution to be expanded as "$AA", "$AB", "$AC", ... A binary zero value will eliminate any prefix character once invoked. For example,

```
*PREFIX  Ø
```

will disengage the MACRO string substitution prefix character.

For more information on the use of the MACRO prefix character, see the chapter on the MACRO PROCESSOR.

*SEARCH filespec

This directive is used to invoke an automatic search of a Partitioned Data Set (PDS) source library. Its syntax is:

*SEARCH filespec	
filespec	Invokes an automatic search of the PDS "filespec/LIB" to resolve any undefined references capable of being resolved by PDS assembler source member modules.

This assembler "*SEARCH filespec" directive is a very powerful feature. It will invoke, a directory search of the Partitioned Data Set "filename/LIB" for all members that will resolve undefined references in the source assembly. This provides a source library structure for EDAS. "*SEARCH" will require two (2) levels of "*GET" nesting. Also, restrictions prevent a "*SEARCH" member from using a "*GET" directive or another "*SEARCH" directive (such a request would be ridiculous anyway). The library members must be lowest level. The default file extension for searched files is "LIB".

The PDS source library constitutes members composed of one or more routines. Each routine that needs to be automatically fetched should have its routine name (the label field entry) in the PDS member directory. This is accomplished by naming the source file to be appended to the library the same name as the routine or by appending using a MAP. Details on constructing and using Partitioned Data Sets is included with PDS documentation. The PDS utility is available separately.

EDAS will search the PDS library and locate a member name that matches up with a symbol table entry. If that symbol is currently undefined, the source member will be accessed and read just as if it were the target of a "GET". EDAS will verify that the member just accessed did in fact define the symbol invoking its access. If a member is accessed and there exists no symbolic label in the member that has the same name as the member name, EDAS will abort the assembly and advise of a library error by displaying the message:

```
Member definition error: filespec(member)
```

At the conclusion of the member's source code, EDAS will continue to search the PDS library until it exhausts all PDS members. There are no restrictions on the order of members. Routines in one member can reference other members with complete disregard as to any ordering of entries in the PDS. EDAS will correctly access all members required.

Where more than one routine is in a member, each should be surrounded by IFREF's/ENDIF and each should have an entry in the member directory (you must use the MAP option of PDS to provide multiple entries to a member). This will benefit by not having needless routines appear in your object code output. For example, the following depicts two routines stored as one member in a PDS.

```
; Entry for routine entitled "MOVE"
        IFREF  MOVE
MOVE    .              ;Routine of code
        .
        .
        ENDIF
; Entry for routine entitled "SHIFT"
        IFREF  SHIFT
SHIFT   .              ;Routine of code
        .
        .
        ENDIF
```

If your source code references "SHIFT" but not "MOVE", as long as both "SHIFT" and "MOVE" are member entries in the library PDS directory, a "*SEARCH" of the library will access the member and assemble only the "SHIFT" routine. You should read the section on the "IFREF" conditional in the chapter on ASSEMBLER PSEUDO-OPS to understand the evaluation of the "IFREF".

WHAT IS A MACRO?

In virtually all programs, you will find particular sequences of code that are repeated. These sequences might be termed short routines. They could be so short that the overhead needed to set them up as CALLable routines is ineffective. Or, they could be longer routines that just cannot be constructed as CALLable segments. You may even want a code sequence to be an in-line assembly in contrast to a CALLable routine for the purpose of fast execution. By far the most needed function, is to be able to have parameterized routines - algorithms that operate on different values each time the algorithm is invoked.

There are at least three ways to deal with routines that are repeated in a program. You can <I>nsert the entire routine wherever it is needed. You could also <C>opy it from the first appearance to wherever you needed the routine. Or you could establish the routine as a macro. The first method is obviously tedious on your fingers. The second, is not tiring, but could take up more source storage than is desirable. Also, if you decide to change the routine's algorithm, having many copies in a program can be cumbersome to update.

The third method mentioned is the use of macros. Consider the following commonplace sequence of code:

```
        LD      HL,VALUE
        LD      (MEMORY),HL
```

How many times is this little sequence repeated in your programs? Five? Ten? If we set up a macro near the beginning of our program that looked something like this:

```
STOR MACRO  #VAL,#MEM       ;Macro to store "VAL" into memory
     LD     HL,#VAL         ;Get value into HL
     LD     (#MEM),HL       ;Load value into memory
     ENDM                   ;End of the macro
```

then we could perform the above two statements with one macro call as follows:

```
        STOR    VALUE,MEMORY  ;Invoke the macro
```

The first part of the example, defines a macro called "STOR". This is done exactly once per program! If we save our macros in a macro source file, each of our programs could "*GET MACROS"; thus, we would not have to even manually enter the macro into each program.

We invoke the statements defined in the macro by specifying the macro name AS IF IT WERE AN OPCODE. Using the macro invocation method, we can save storage space and introduce structured techniques to our coding. Notice that we have used some fictitious names when the STOR macro was defined. These names are called "dummy" parameters. They serve to provide a means to pass

actual parameters when the macro is invoked. It is through the dummy parameters that the real power of the macro is utilized. During the macro invocation, the model statements are expanded with substitutions for the dummy parameters that are provided in the macro call.

MACRO DEFINITION

The format for a macro definition is illustrated in the following example:

```
MOVE    MACRO   #parm1,#parm2=dflt2,#parm3
        LD      HL,#parm1
        LD      DE,#parm2
        LD      BC,#parm3
        LDIR
        ENDM
```

The macro definition consists of three parts: a macro prototype, a macro model, and the ENDM statement. The prototype is used to specify the macro name and the dummy parameter names used in the model. Default substitutions may be specified in the prototype to be used if the corresponding parameter is not passed in the macro invocation. The macro model contains all of the assembler statements to be generated when the macro is invoked. The model is sometimes called the macro skeleton or template. The dummy parameter names occupy the positions where the actual parameters will be placed by the macro processor in EDAS. The third part, the ENDM statement, is used to indicate the end of the macro model.

When a macro is defined, it is not assembled into your program. The macro prototype is parsed and analyzed. The macro definition is then stored in a compressed format within the macro storage area. Comments appearing with the macro definition are not stored. That means that if the macro expansions are listed in the assembler listings, they will not include the comments - only the definition will.

Macro Prototype

The MACRO pseudo-OP is used to define the prototype of a macro model. Its syntax is:

```
mname  MACRO  {#parm1}{=dflt1}{,#parm2{=dflt2}}{,...}

mname       is the macro name used to invoke the macro.

#parmn      are dummy parameters of the macro which will
            be replaced by actual parameters during the
            macro invocation. "#" is a required prefix.

dfltn       are optional default strings to be used for
            the dummy parameters when a parameter is not
            provided in the macro invocation.
```

Macros are named just like symbolic labels. The same rules apply. The length of macro names can range from <1-15>. Special characters <@, $, _> may be used in the name construct. Do not use the question mark in macro names as it would conflict with the symbol substitution string use made of "?".

There is no upper limit on the number of macro parameters; however, you can not exceed the length of a standard assembler source statement. Therefore, the statement length becomes the limiting factor. As is the case with macro names, the rules for naming dummy parameters are identical to the rules for labels. The "dummy" names are not included in the symbol table generated by EDAS, thus there is no restriction on reusing the same name as a "dummy" for a label; however, to avoid confusion, it is recommended that you avoid using dummy names as symbolic label names.

Default strings can contain any character except the comma, ",". The comma is used as a field delimiter. There is no limit to the length of a default string other than the limiting factor of the statement length.

Macros must be defined prior to use but can be defined in either disk "*GET files" or memory text.

Macro Model

Any valid Z-8Ø statement, EDAS pseudo-OP, or assembler directive (except "*GET" or "*SEARCH") is valid in the macro model - except the "MACRO" pseudo-OP (no nested definitions, please).

ENDM pseudo-OP

This pseudo-OP is used to specify the scope of a macro model. It is used much like the "ENDIF". Its syntax is:

```
mname  MACRO  parms
       model statements
       ENDM
```

The "ENDM" pseudo-OP must be used to let the macro processor know what is the last macro model statement.

Macro Definition Examples

This macro will move a block of memory from one location to another. If the "length" parameter is omitted, then a value of "255" will be used:

```
MOVBLK  MACRO  #FM,#TO,#LEN=255
        LD     HL,#FM
        LD     DE,#TO
        LD     BC,#LEN
        LDIR
        ENDM
```

This is a macro to clear a region of memory (i.e. set to Ø). This macro will invoke the MOVBLK macro in a nested invocation:

```
CLRMEM  MACRO  #BUF,#LEN=255
        LD     HL,#BUF
        LD     (HL),Ø
        MOVBLK #BUF,#BUF+1,#LEN
        ENDM
```

This macro will add the 8-bit register "A" to 16-bit register pair "HL":

```
ADDHLA  MACRO
        ADD    A,L
        LD     L,A
        ADC    A,H
        SUB    L
        LD     H,A
        ENDM
```

There is no requirement that a macro must contain dummy parameters as is evidenced by the last example.

Incorporating Conditionals

Conditional pseudo-OPs can be specified in macro models. For instance, say you want the MOVBLK macro to be able to perform a non-destructive move (a destructive move would be where the destination is an address between "from" and "from+length-1"). You can insert conditional pseudo-OPs to test the parameters during the assembly of the expansion (labels substituted for #FM and #TO must be defined prior to invoking the MACRO). Then, only certain segments of the macro will be assembled according to the result of the evaluation. Analyze the following example:

```
MOVBLK MACRO   #FM,#TO,#LEN=255
       IFNE    #FM,#TO          ;Don't expand if #FM=#TO
       LD      BC,#LEN          ;Establish the length
       IFGT    #FM,#TO          ;Do we LDIR or LDDR?
       LD      HL,#FM           ;#FM > #TO => LDIR
       LD      DE,#TO
       LDIR
       ELSE
       LD      HL,#FM+#LEN-1    ;#TO > #FM => LDDR
       LD      DE,#TO+#LEN-1
       LDDR
       ENDIF
       ENDIF
       ENDM
```

MACRO NESTING

The CLRMEM example depicts a macro that nests a macro invocation. Macros may be nested to seven (7) levels. That is, at any time, macro expansions for 7 macros called in a chain can be pending. It is very important to note that macro definitions cannot be nested. For instance:

```
ABC    MACRO     #PARM
       (model statements)
XYZ    MACRO     #PARMs,...
       (model statements)
       ENDM
       ENDM
```

is illegal and will result in an assembly error. It is entirely correct, however, to invoke a macro within a macro definition prior to the definition of the called macro. The called macro must, however, be defined prior to calling the first, or highest level, macro. For example:

```
ABC   MACRO    #PARMS,...
      (model statements)
      MOVE     parm,parm   ;call macro "MOVE"
      (model statements)
      ENDM
MOVE  MACRO    #parm1,#parm2,#parm3
      (model statements)
      ENDM
```

is perfectly legal. The expansion of the "MOVE" macro is not performed during the definition of the "ABC" macro but rather during the invocation of "ABC".

If macro A "calls" another macro, say B, any dummy parameter in the macro call of B that matches a dummy in macro A, will be considered part of macro A and the parameter substitution will be invoked by the parameter passed when the user calls macro A.

MACRO INVOCATION

The invocation of a macro is termed a macro "call". The macro processor then proceeds to replace the call with the model statements specified when the macro was defined. The replacement of the macro call by the macro model statements is termed the macro "expansion".

During the expansion, the "actual" parameters passed in the call statement are substituted for the "dummy" parameters which appear in the macro model and which are designated in the prototype of the macro. Note that the actual parameter values are character strings and can be labels, expressions, or data constants. An actual parameter can even be a quoted string data declaration if its use is designed into the macro model.

The entire expanded macro model is listed during the listing pass (phase two) of EDAS. You may find that you don't really want to see these expansions since the macro definition contains the entire illustration of the macro. An assembler switch, "-NM" is provided in the <A>ssemble command to suppress listing of macro expansions. In the case of nested macro calls (i.e. a macro is defined which calls another macro which was separately defined), only the primary macro call will be listed if the "suppress" switch is invoked.

The substitution of the actual character string parameters for the dummys occurs during the macro expansion when the macro is called. Since a macro can have more than one parameter, it is necessary to have a procedure that specifies which actual parameter corresponds to each dummy parameter. There are two methods supported in EDAS. Parameters can be passed to the macro expansion when calling by either position or keyword.

Positional Parameters

"Positional" parameters are correlated by the position they appear in the macro call. For example, if the "MOVBLK" macro were called by the statement:

```
MOVBLK  VIDEO,CRT_BUFFER,CRT_SIZE
```

then the substitution string "VIDEO" would replace every appearance of "#FM", the string "CRT_BUFFER" would replace every appearance of "#TO", and "CRT_SIZE" would replace the dummy parameter, "#LEN". Note that actual strings are positionally correlated with the positions of the dummy parameters in the macro prototype.

If you wish to omit an actual parameter in a macro call, then you must supply the comma to denote its place. For instance:

```
SHIFT  4200H,,100H
```

omits the middle of three parameters. Generally, a default would have been provided in the macro definition.

Keyword Parameters

If the number of parameters is large, it is sometimes burdensome to remember the order of the parameters, or to provide the correct number of commas if a series of parameters are omitted. These drawbacks are remedied by the use of "keyword" parameters. The macro call parameter list can identify the actual parameters by using the name of the dummy parameter as well. The keyword syntax is:

```
        #dummy=actual parameter

mname   #parm2=actual2,#parm3=actual3
```

If the previous macro call was invoked by keyword parameter specification, it could look something like this:

```
SHIFT  #LEN=100H,#FM=4200H
```

Mixing Positional and Keyword Parameters

A single macro invocation can intermix both positional and keyword parameters. The point that needs clarification, is what positions are actually denoted in the parameter list. It is simply treated. In a mixed parameter list, keyword parameters are ignored when considering place positions. For example, in the following macro call:

```
SHIFT  #LEN=1ØØ,BLOCK,BUF_START
```

even though the length parameter appeared first in the parameter list, since it was designated as a keyword, it is ignored from the positional count and "BLOCK" is the first parameter with "BUF_START" taking up second place. In a similar manner:

```
COMP   PARM1,#P6=2,,PARM3,#P8=38,PARM4
```

"PARM1" is in position one, the second parameter is omitted (the double comma), "PARM3" and PARM4" are in the third and fourth positions respectively. The sixth and eighth parameters have been entered by keyword.

Please note that the parameter list contains five parameters. Thus if you were to use the "%%" operator which returns the number of parameters passed in a macro call ("%%" is described later), it would return a value of five.

LOCAL LABELS

So far, all of the examples have shown macro models without labels. What would happen if we had a macro defined as follows:

```
FILL MACRO  #CHAR,#NUM
     LD     B,#NUM
FLP  LD     (HL),#CHAR
     INC    HL
     DJNZ   FLP
     ENDM
```

We would have a problem because every time the macro was called, the label, "FLP", would be used. If "FILL" was invoked more than once, the assembler would generate MULTIPLY DEFINED SYMBOL errors on each expansion. We have to be able to use labels, but we need to find a way to be able to make "unique" labels on each macro expansion.

EDAS provides a facility for doing this by keeping a substitution string which is changed each time a macro is expanded - any macro. The substitution string replaces the question mark character, "?", during the macro expansion whenever it appears outside of single quotes in a macro model statement. Each time a macro is expanded, the "value" of the string will be changed. The

"value" starts with the single letter "A", changes to "B", ..., "Z", then increments to the two-letter strings, "AA", and changes to "AB", "AC", ..., "BA", ..., "ZZ" each time a macro call is made. Thus, by incorporating the question mark as one of the characters in the label of a macro model statement, it can be used to uniquely identify labels local to a macro. You may want to standardize the way you create labels to ensure that uniqueness is maintained. For example, if you use macro labels of the form, "$$?1", "$$?2", ..., these will expand to "$$AA1", "$$AA2", ... within one macro during its first expansion. The second macro expansion will create "$$AB1", "$$AB2", ... You can then repeat the use of "$$?1", "$$?2", ..., in another macro since for each macro expansion, the substituted string will be different.

The substitution string will be different from the "*MOD" directive substitution but is similarly used. Macro expansion substitution of "?" takes precedence over *MOD substitution. In the case of nested macros, each nest level will have its own unique substitution (since each nest is a macro call which invokes an expansion).

The macro substitution string can be prefixed with a user-entered character constant. This is achieved by using the "*PREFIX" assembler directive as in:

```
*PREFIX character-expression
```

where the expression character or value in the argument becomes the prefix character (it must be valid for assembler source labels). For example, "*PREFIX '$'" will cause macro local label string substitution to be expanded as "$AA", "$AB", "$AC", ... A binary zero value will eliminate any prefix character once invoked.

By using the question mark string substitution specifier, the previous macro would be defined like this:

```
FILL MACRO  #CHAR,#NUM
     LD     B,#NUM
$$?1 LD     (HL),#CHAR
     INC    HL
     DJNZ   $$?1
     ENDM
```

STRING COMPARISONS

It is sometimes desirable to be able to test within a macro model, the exact string passed as a parameter. Four conditional pseudo-OPs have been added strictly for string comparisons within macro processing. These are:

IFLT$	string1,string2	TRUE if string1 < string2
IFEQ$	string1,string2	TRUE if string1 = string2
IFGT$	string1,string2	TRUE if string1 > string2
IFNE$	string1,string2	TRUE if string1 <> string2

These pseudo-OPs provide TRUE/FALSE evaluation in the comparison of string1 to string2 (like the non-"$" pseudo-OPs do with mathematical expressions). Obviously, hard encoding of both string1 and string2 would be nonsense! Aha, he said... If we use a macro dummy parameter, it will be substituted by the actual parameter string passed in the macro call expansion. This means that the macro itself can test the parameter string in a limited manner. For example:

```
IFNE$   #TO,(DE)
LD      DE,#TO
ENDIF
```

as part of a macro model, will have the "#TO" replaced during the expansion. The test becomes dynamic! The dummy parameter can be either string1 or string2 - it doesn't matter.

These string conditional pseudo-OPs can only be useful in macros. That's because the evaluation, to make sense, has to be dynamic.

TESTING STRING LENGTHS

Another feature available in the macro processor is the per cent sign "%" operator. This operator is used to recover the length of the passed parameter string and the number of parameters passed in the macro call. Note that the limitation for the use of the "%" operator, is that it is acceptable only for parameters of the current macro expansion. That means that you can't test for lengths outside of the current macro if you are nesting macro calls (macros cannot be recursive!). The operator can be used like these examples:

```
LD   B,%#PARM          ;loads B with the length of #PARM

IFGT %#PARM1,6         ;Restricts parm1 to a length <1-6>
ERR  Parm too long!
ENDIF

IFLT  %%,4             ;This macro requires 4 actual parms
ERR   Missing required parameters!
ENDIF
```

As can be noted, the "%%" operator will return the number of parameters passed in the current Macro call. When a dummy parameter name (including the "#" prefix) follows the per cent operator, the length of the parameter string is returned.

These values can be tested arithmetically to produce a TRUE/FALSE result (as was just demonstrated), or they can be used directly to represent logic TRUE/FALSE conditions. Realizing that if a parameter was not passed in the parameter list of the macro call, its length would be zero. A zero is also a logical FALSE. EDAS will accept as TRUE, any non-zero value (in normal use of TRUE/FALSE specifications, "-1" is recommended for TRUE to maintain proper evaluation of the ".NOT." operation). Thus, the string lengths can be minimally used to test if the parameter was not passed (%#parm=Ø=FALSE) or the parameter was passed (%#parm<>Ø=TRUE).

CONCATENATING MACRO LABELS

You can concatenate a string to a dummy parameter name by connecting it with the concatenation operator, "%&". For instance, the model statement:

```
IFREF    #NAME%&L
```

will have the "#NAME" replaced by the MACRO call substitution string appended with the letter "L".

Editor Assembler Commands

The EDAS Version IV Editor Assembler can perform the following commands. These commands may be typed after the prompt symbol ">". The prompt symbol appearance indicates the "command mode" of the Editor Assembler. The following list contains all command mode instructions recognized by the Editor Assembler with a brief description of each.

A <A>ssemble source currently in the text buffer.

B <B>ranch to a specified address.

C Globally <C>hange a string of characters (STRING1) to another string of characters (STRING2) throughout a range of text lines.

C <C>opy a block of lines to another location.

D <D>elete specified line(s).

E <E>dit a specified line of text.

F <F>ind a specified string of characters.

H Provide <H>ard copy output (line printer) of a specified range of text buffer lines.

I <I>nsert source text line(s) at a specified line with a specified line number increment.

K <K>ill a file from a diskette.

L <L>oad a source text file from disk.

M <M>ove a block of text from one location to another.

N Re<N>umber source text lines in the text buffer.

P <P>rint a specified range of source text code currently in the text buffer.

Q <Q>uery a directory from the designated drive.

R <R>eplace lines currently in the text buffer.

S <S>witch the upper case/lower case conversion mode.

T <T>ype source text lines without line numbers to a line printer.

U Display the memory <U>tilization - bytes used by the text, bytes available, and the first free address.

V <V>iew a file without loading it into the text buffer.

W Write the current text buffer to disk.

X eXtend the text buffer by eliminating the Assembler.

Z Command reserved for user.

1 Alter printed lines per page and page length.

. Send a message to a Job Log (LDOS only).

CLEAR Clear the CRT screen.

UPARW Scroll up one source text line.

DNARW Scroll down one source text line.

LTARW BACKSPACE key

RTARW TAB key

SRARW Page forward one screen.

PAUSE Performs a functional pause of any operation: <SHIFT @ (Model I/III)> <HOLD (Model II)> for the PAUSE function).

UPARW => the up-arrow key
DNARW => the down-arrow key
LTARW => the left-arrow key
RTARW => the right-arrow key
SRARW => the shifted right arrow key (F2 on Model II)

<A>SSEMBLE

The <A>ssemble command is used to invoke the assembly of your source stream from memory and optionally, disk files (when "*GET filespec" or "*SEARCH library" is used in the source stream). The <A>ssemble command is also used to create a cross reference data file for downstream processing by the XREF/CMD program which will create a complete symbol cross reference listing. The syntax of the <A>ssemble command is:

A {filespec1/CMD}{,filespec2/REF} {-SWITCH {-SWITCH}...}

filespec1	is the filespec to be used for the object code file generation. If the file extension is omitted, "/CMD" will be used (see -CI).
filespec2	is the filespec to be used for the cross reference data file. If the file extension is omitted, "/REF" will be used.
Switches:	
-CI	used to generate a Core-Image object file.
-IM	used to assemble the object code Into Memory.
-LP	used to generate a Listing to the Printer.
-NC	used to suppress the listing of conditional blocks evaluated to be logically FALSE.
-NE	used to suppress the listing expansion of data declaration pseudo-OPs.
-NH	used to suppress writing the header record to the object code file.
-NL	used to suppress the listing pass.
-NM	used to suppress listing MACRO expansions.
-NO	a dummy switch useful as a default switch in JCL execution of EDAS.
-SL	used to suppress local label listing
-WE	used to pause the assembly listing and Wait if an Error occurred.

Parameters continued next page

-WO	used to assemble With Object code generation.
-WS	used to generate a sorted symbol table listing during the assembly process.
-XR	used to generate a cross reference data file for subsequent processing by XREF/CMD.

The <A>ssemble command can be used to generate object code into either an executable object code file (/CMD) or a binary core-image object code file (/CIM). Your program can also be assembled directly into the unoccupied memory region when the memory locations to be occupied by your program are not in conflict with storage areas of the assembler, your resident source code, the MACRO storage area, or the symbol table.

The source text to be assembled can exist either in memory only, or a combination of memory and disk files. The in-memory source is considered to be in the "text-buffer". When your source program is too large to be contained solely in the text buffer, it needs to be segmented into a combination of a memory segment and one or more disk file segments. The disk file segments are accessed during the assembly process by use of the "*GET filespec" assembler directive (detailed instructions concerning the use of *GET, are contained in the chapter entitled "ASSEMBLER DIRECTIVES").

The following paragraphs describe the command line entries and switch options in detail. Please note that if the EDAS e<X>tend command has been invoked, the <A>ssemble command will be inoperative.

Filespec1

The first filespec on the command line, identified as "filespec1", is the filespec to be used for the object code file. Its entry is entirely optional. When an object code filespec is entered, its entry will automatically invoke the generation of the object code to the disk file. Another method can also be employed to invoke object code generation to a disk file by means of the "-WO" switch (see below). If your filespec entry omits the file extension, the default of "/CMD" will be used. This default is changed to "/CIM" if the "-CI" switch is specified. It is recommended that you let the assembler assign the file extension, automatically. It will help to keep your directories orderly, and there will be less danger of overwriting a source file with the object code file.

Filespec2

The second filespec on the command line, noted as "filespec2", identifies the filespec to be used when writing the cross-reference data. The

cross-reference data generation is optional - it is required in order to run the XREF/CMD program. EDAS will assign a default file extension of "/REF" if you omit the extension from your filespec. As XREF/CMD will also use this extension when accepting the file specification, it is suggested that you let EDAS assign it. You can also invoke generation of cross-reference data by using the "-XR" switch (see below). EDAS requires the entry of the comma to recognize the cross-reference filespec as "filespec2". Therefore, if you want the cross-reference data file but not the object deck file, then either start the command line with the comma separator or use the XR switch without entering the filepec with the command line.

Switch -CI

The "-CI" switch is used to generate a "core-image" object code file. Executable command files in LDOS are constructed with address information that the system loader uses when loading and executing your command file. Also, a header record is usually found in a load module object code file. There are times when you would prefer an object code file without this "load" and "comment" data. For example, say you want to burn a Programmable Read Only Memory (PROM) from a file. A core-image file is needed. When the "-CI" switch is specified, a number of changes take place in EDAS. First, the object code file default extension is changed to "/CIM" (note: you must still enter the filespec or the switch "-WO" to invoke object code generation). Next, the header record and the transfer address record are suppressed. Any COM pseudo-OP statement is, likewise, suppressed. A core-image file needs to contain contiguous address sequential code. Since EDAS reserves only storage locations when assembling the DS/DEFS pseudo-OPs, the DS instruction would cause your object code file to be non-contiguous. Invoking the "-CI" will automatically convert all "DS" statements to their corresponding "DC" statements with a zero value for operand2.

Switch -IM

This switch will invoke object code generation; however, instead of the code being written to a file, it is placed into memory starting at the address specified as the operand of the "ORG" pseudo-OP. The "-IM" switch will override the entry of the "-WO" switch or entry of "filespec1". That is, if both "-IM" and "-WO" (or filespec1) are entered, assembly into memory will occur and assembly to disk will NOT take place.

Your program will not be permitted to overwrite any region below the end of the text buffer (or macro storage area if macros are being used) nor will it be permitted to overwrite the symbol table stored in high memory. The error message,

Memory overlay aborted

will be displayed if your assembled program will violate these restrictions. The assembly will be immediately stopped and EDAS will return to the command ready prompt. Upon successful completion of the assembly to memory, the

message,

```
Memory region loaded
XXXX is the transfer address
```

will be displayed. This does not mean that your program assembled without error - only that the object code generated did not interfere with the text buffer or tables created during the assembly process. The "XXXX" field in the second message will contain the transfer address of the program. It will be listed in hexadecimal.

Switch -LP

The "-LP" switch is used to send the assembler listing, error messages occurring during the assembly of your source code, and the symbol table listing (if specified by means of the "-WS" switch) to a line printer. EDAS assembler listings print 56 lines per page and send a form feed at the conclusion of the 56 lines. If you are generating a listing output and a properly paged display is desired, it is suggested that you set your paper to begin printing at the sixth line from the top of the page (which assumes paging parameters set at 56 print lines and 66 lines page length - the default). This will provide five blank lines for a top margin, and five blank lines for a bottom margin.

If you are using other than 11" form paper, use the EDAS command "<1>" to alter the paging parameters to suit the specifications of your printer.

Switch -NC

Conditional assembly (see the chapter on ASSEMBLER PSEUDO-OPS) can greatly ease the maintenance of programs designed to work with multiple configurations of hardware. However, it is unnecessary to "see" the source statements within conditional blocks that are logically "false". This "-NC" switch is provided to have No "false" Conditionals appear in your listings. If a conditional is suppressed, neither the "IF" statement nor the "ENDIF" statement of the "false" block will be listed.

Switch -NE

Various data declaration pseudo-OPs create a structured format for the listing of code generated after the first byte of the statement. These are the DB/DEFB, DM/DEFM, DW/DEFW, and the DC pseudo-OP statements. If you want to inhibit the expansion from the listing only (the code will still be expanded for assembly of object code), then specify the No Expansion, "-NE", switch.

Switch -NH

Object code files usually start off with a header record of X'Ø5 Ø6 xx xx xx xx xx xx'. The x's would be replaced with the first six characters of the object code filename (buffered with spaces). EDAS automatically generates this record when writing the object code file. The DOS loader has no problem with this record. If you would like your object code files to contain this record, then do absolutely nothing. If you do not want to have this header record generated, then specify the No Header, "-NH", switch.

Switch -NL

The second phase of the assembly process generates the assembler listing. That is the only purpose it serves. If you do not want to see a listing, then you may enter the No Listing, "-NL", switch. This will completely suppress phase two and shift the assembler to phase three (if object code generation had been specified. If you are interested in listing statements containing errors, then you must not suppress the second phase. Note that only the lines containing assembly errors can be listed by specifying the "*LIST OFF" assembler directive. See the chapter on ASSEMBLER DIRECTIVES" for further details.

The cross-reference data file is written during phase two. In order to guarantee that the second phase is available, a cross-reference specification will automatically override any entry of the "-NL" switch. This could be useful during a job stream assembly (from Job Control Language) where selected assemblies need the cross-reference data. Thus, your JCL could specify "-NL" for every assembly; whenever the XR option was invoked, phase two would not be suppressed.

Switch -NM

You have read about the powerful uses made of macros in the MACRO PROCESSOR chapter. By now, you may have realized that the macro model code is repeated whenever you invoke the macro. Once you become familiar with what the macro does, you really don't need to see its expansion in your listings every time the macro is invoked. Switch "-NM" has been provided to inhibit the listing of such expansions. If you specify No Macro expansions, only the statements invoking the macros will be listed - the listing of the expansions will be inhibited. In the case of a nested macro invocation, only the highest level macro call will be listed.

Switch -NO

Previous versions of EDAS, and other assemblers (are there any other?) have used a switch designated "-NO". Its use was to inhibit the generation of object code (No Object) when the assembler automatically generated the object code. Since EDAS does NOT generate object code unless you tell it to do so

(by "filespec1", switch "-WO", or switch "-IM"), the "-NO" switch is unneeded. There are those "old dogs" that cannot learn new tricks. Therefore, switch "-NO" has been included just in case you have the habit of entering, -NO. However, it does absolutely NOTHING!

An alternate use can be made of the "-NO" switch when operating EDAS from Job Control Language. This was addressed in the chapter entitled, RUNNING EDAS.

Switch -SL

If you specify "-SL", then any label starting with a dollar sign, "$", will be suppressed from the symbol table listing and from any cross-reference data file. Therefore, use of the "$" as the first character of local labels and specifying "-SL" will result in keeping your symbol table listings uncluttered with local labels - especially true with the LC compiler.

Switch -WE

In a long assembly, you may want the assembler to pause the listing if it detects an assembly error (you're bound to get some of them). The Wait on Error switch, "-WE", is available for that purpose. If specified, each time the assembler comes to an error during phase two, it will pause the listing. Any character entered from the keyboard will continue the assembly and listing. If you choose to enter the character "C" or "c", then the phase two process will "c"ontinue without further interruption - even though additional errors may be detected. The listing may also be paused at any time by depressing the <PAUSE> key, momentarily.

Switch -WO

As noted in a preceding paragraph, object code generation is specified when "filespec1" is entered. Assembled object code is also generated to disk if the With Object switch, "-WO" is specified. If "filespec1" has not been entered, the prompt message:

Obj filespec?

will be displayed. Enter the object code filespec that you want to use to save the assembled object code command file at this time. If you do not enter a file extension, the default "/CMD" will be assummed. EDAS will open the file if it is an existing file and display the message, Replaced, or create the file if it is non-existant and display the message, New file.

If you enter "filespec1", it is not necessary to enter the "-WO" switch as entering the object code filespec will activate the "-WO" switch. If the switch, "-IM", is specified denoting an in-memory assembly, the "-WO" switch will be ignored.

Switch -WS

A complete symbol table cross-reference listing is available via the "-XR" switch and subsequent processing by the XREF/CMD program. Such a separate process is needed in order to be able to handle cross referencing of statements fetched from a *GET or *SEARCH file. An abbreviated printout that contains only a sorted listing of symbols and their value is available at assembly time by invoking the With Symbol switch, "-WS". The symbol table listing would normally be displayed on the video display. If the "-LP" switch was specified, the listing would be directed to the Line Printer.

Switch -XR

This is the switch option to use if you want to generate a complete symbolic cross reference listing. Switch "-XR" will invoke the generation of a reference data file used by the XREF/CMD utility (see the chapter on CROSS REFERENCE UTILITY). The reference data file is generated during the listing pass (phase two). If the XREF filespec is entered with the command line, this switch is assumed to have been entered. If the XREF filespec is not entered with the command line, the filespec of the reference file will be prompted for with the query,

XREF Filespec?

Respond with the filespec that you want to use to store the reference data. If you do not enter a file extension, the default "/REF" will be assumed. EDAS will open the file if it is an existing file and display the message,

Replaced

or create the file if if it is non-existant and display the message,

New file

Error totals

At the conclusion of phase three which generates object code, a listing of the total number of errors will appear. This error total will be displayed after the conclusion of phase two if object code is not generated. If you need to get a quick idea whether or not your source code contains errors, place an "*LIST OFF" pseudo-OP at the beginning of your code and omit any object code generation - but do not specify "-NL". Only lines containing errors will be listed. You could also specify switch "-WE" to pause when an error occurs [Note: If you specify -NL and do not generate object code, the "Error totals" will be incorrect (the number of forward references plus any other errors will be displayed)].

<B>RANCH

The <B>ranch command is used to exit EDAS. Since the <B>ranch command permits an address as an optional parameter, you can use it to jump to any address (the entry to an in-memory assembled program, for instance). The syntax of <B>ranch is:

B {address}
address is the branch address entered in hexadecimal.

This command is used to exit the Editor Assembler or optionally branch to any user designated address. If a branch address is omitted, a return to the DOS Ready command mode is performed. If a branch address is provided, the top of the stack will contain a re-entry address to EDAS. This can benefit the testing of a program assembled into memory. A simple "RET" instruction in your program will return control to EDAS (provided your program maintained stack integrity and did not crash).

Examples of the <B>ranch command:

B

"B" by itself will cause an exit from EDAS and return to DOS

B 9000

This command will cause an exit from EDAS and branch to your program at X'9000' (it is hoped that your program is there).

B 5806 (Model I/III) or B 3706 (Model II) or B3606 (LDOS 6.x)

This will invoke a jump to the warmer-start vector which re-initializes EDAS and clears the text buffer.

B 30

This branch will cause EDAS to enter DEBUG (Model I/III or LDOS 6.x only). The Program Counter as displayed by DEBUG can be used as the return address to EDAS. Optionally, you can "Go" to X'5803' (Model I/III) or X'3603' (LDOS 6.x) or X'3703' (Model II).

<C>HANGE

The <C>hange command performs a global modification of a string of characters. Its syntax is:

C /string1/string2{/n1,n2}	
string1	is the current string to change.
string2	is the replacement string for string1.
n1	is the line number of the line preceding the first change (FIND always starts at line+1).
n2	is the line number of the last line to change.
/	represents a string separator character. It can be any character except a digit <Ø-9>.

A string of characters can be changed throughout the text buffer by this one easy command. The global <C>hange command will change the appearances of "string1" to the sequence "string2". Because <C>hange uses the <F>ind command to locate strings and the <F>ind command always starts searching at "current line + 1", no changes can be performed on the first line of the text buffer - at least not with the <C>hange command. Also, only the first appearance of "string1" in each line that "string1" appears will be altered.

The first non-blank character following the "C" becomes the string delimiter (the slash character is shown above; any character except a digit <Ø-9> is permitted). Null strings are not permitted (i.e. the string must contain at least one character).

There is no requirement for "string2" to be the same length as "string1". It can be of lesser, equal, or greater length; however, no string can exceed 16 characters in length. If a change would result in a line exceeding the maximum line length (which is 128), the change will not be performed on that line and the message,

Field overflow

will be issued. The search for "string1" continues for the remaining lines.

A line which contains "string1" will be displayed as it exists both before and after the change. The <SHIFT-@> key may be used to pause the display. If you depress the <BREAK> key, it will stop further changing.

The entry of "n1" and "n2" is optional. If "n1" is entered, then "n2" must be entered. If neither "n1" nor "n2" is entered, then "n1" is assumed to be the beginning of the text buffer (# or t) and "n2" is assumed to be the end of the text buffer (* or b). Either "n1" or "n2" can be entered as the current line indicator (.). You can enter "n1" as (# or t) to indicate the beginning or top of the text buffer while "n2" can be entered as (* or b) to indicate the bottom of the text buffer. One additional restriction is that if you enter "n2" as "b" or "*", then no change will be made on the last line of the text.

When EDAS is set to the "lower-case converted" mode (see the information concerning the "<S>witch-case" command), both "string1" and "string2" will be converted to upper case characters prior to the search and replacement. If you need to change lower case characters as well, then you must switch EDAS to the "lower-case permitted" mode prior to issuing the <C>hange command.

The "tab" character is a perfectly acceptable character to be used within "string1" or "string2". This may be useful if you want to convert a contiguous sequence of spaces to a single tab.

Examples of the <C>hange command:

C /MODIFY/ALTER/

This command will change all appearances of the string "MODIFY" to the string "ALTER".

C .DEFB.DB.90,1000

This command will change all appearances of "DEFB" to "DB" from line 100 to line 1000 (assuming inc=10).

C /DEFM/DB/90,b

This <C>hange command will translate all appearances of "DEFM" to "DB" from line 100 to the end of the text (assuming inc=10).

<C>OPY

The <C>opy command can be used to duplicate a line or block of lines from one point in the text buffer into another point in the text buffer. Its syntax is:

C line1,line2,line3	
line1	is the first line of the block to duplicate.
line2	is the last line of the block to duplicate.
line3	is the line number of the line that the copied block should follow.

This command is useful to duplicate a line or block of lines. Note that the command letter is the same as the <C>hange command. EDAS will interpret the <C> as a <C>opy command if the first non-blank character following the <C> is a digit <Ø-9>. At the conclusion of the <C>opy operation, the entire text will be renumbered using the increment currently in effect. A few restrictions are in order. A <C>opy cannot be performed if "line3" is interior to the block "line1"-"line2". "Line1" must either precede "line2" or be equal to "line2" (where "line1" is equal to "line2", the block to be duplicated consists of the single line, "line1").

If insufficient space is remaining in the text buffer to duplicate the entire block, none of the block of lines will be copied and the message,

Text buffer full

will be displayed. The parameters (line numbers) must specify specific lines in the text buffer. If any of the line numbers cannot be found, the copy will not be performed and the message,

No such line

will be displayed. The <C>opy command requires all three parameters entered and separated with the comma (,). If this syntax is not met, the message,

Bad parameters

will be displayed.

Examples of the <C>opy command:

C 100,200,1000

This command will duplicate the block of lines numbered from 100 to 200 inclusive to also appear after line number 1000.

C t,50,50

This command will copy the block of lines from the top of the text through line number 50 so that it will also follow line number 50.

c 580,700,b

This <C>opy command will duplicate the block of lines numbered from 580 to 700 so that they also appear after the current bottom of text.

<D>ELETE

The <D>elete command is used to remove a line or block of lines from the text buffer. Its syntax is:

```
D {line1{,line2}}

line1        is the first line to delete.

line2        is the last line to delete.
```

This command is used to delete the line or lines specified from the source text buffer. The characters "#" or "t" are used to indicate the beginning of the text buffer when used for "line1". The characters "*" or "b" are used to indicate the bottom of the text buffer when used for "line2". If the line parameters are omitted, the current line, "." is assummed.

To aid in you in observing what becomes the new current line after a line delete operation, the new current line will be displayed.

Examples of line deletes:

D 100,500

This <D>elete will remove from the text buffer, lines 100 through 500 (inclusive).

D T,B or d t,b or d #,*

This command will remove the entire source text from the text buffer. A <B>ranch to the "warmer" start address also will delete the entire text.

D or d

This <D>elete command will remove the current source text line. A period, ".", may also be used to indicate the current line (i.e. "D.").

D 105

This command will delete the the single line numbered 105.

<E>DIT

The <E>dit command is used to invoke the line editor for purposes of making alterations to a single text line. Its syntax is:

```
E {line}

line        is the number of the line to edit.
```

This command permits the user to edit or modify any source text line. The syntax and function of all edit subcommands are identical to those implemented in the BASIC editor. If the optional line number is not entered, the current line, ".", will be edited.

When using the line editor, it will always operate in the "lower-case permitted" mode. Therefore, you will need to pay attention to use of the <SHIFT> key when editing upper-case characters. However, once you complete your editing and exit the line editor, your line will be properly converted to upper-case as required if EDAS is in the "lower-case converted" mode.

Editor Assembler Commands

The following table of Edit Subcommands are provided for a reminder of the common edit operations:

A	Abort and restart the line edit.
nC	Change n characters.
nD	Delete n characters.
E	End editing and enter the changes.
H	Delete (hack) the remainder of the line and insert the following string. A line hacked to zero length will be automatically deleted when exiting the line editor.
I	Insert string.
nKx	Kill all characters up to the nth occurrence of x.
L	Print the rest of the line and go back to the starting position of the line.
Q	Quit and ignore all editing.
nSx	Search for the nth occurrence of x.
<--	Move edit pointer back one space.
ENTER	Enter the line in its presently edited form and exit the edit mode.
ESCAPE	Escape from any edit mode subcommand. The <SHIFT-UP-ARROW> key is the escape key on the Model I and Model III.
SPACE	Display the next character of the current line being edited.

<F>IND

The <F>ind command is used to locate the next occurrence of a string of characters within a line. Its syntax is:

F {string}
string is the character sequence to find.

The text buffer is searched starting at the current "line+1" for the first occurrence of "string". "String" can be from <1 to 16> characters in length. If more than 16 are entered, then any characters beyond the 16th will be ignored. If no string is specified, the search is the same as that of the last <F>ind command in which a string was specified (provided a global <C>hange command was not performed after the last <F>ind command). If the search string is found, the line containing it is displayed and the current line pointer, ".", is updated to point to the displayed line. If the string is not found, the message,

String not found

is displayed and the current line pointer, ".", remains unchanged. A "P#" or "Pt" command can be used to position the line pointer to the top of the text buffer prior to use of the <F>ind command. Spaces and tabs are considered to be part of "string" and are thus acceptable for "finding".

Examples of the <F>ind command:

FWRITEWORD

This <F>ind command will locate the next appearance of the string "WRITEWORD".

F

Assuming a <C>hange command has not been performed, this command will find the next appearance of "WRITEWORD".

<H>ARDCOPY

This command lists a line or block of lines on a line printer to provide a "hard copy". Its syntax is:

H {line1{,line2}}	
line1	is the line number of the first line to print.
line2	is the line number of the last line to print.

This command will print a line or a group of lines to a line printer. EDAS will print 56 lines to a page (see the discussion of the <1> command). If a properly paged display is desired, it is suggested that you set your paper to begin printing at the sixth line from the top of the page.

Examples of the <H>ardcopy command:

H #,* or H t,b

This command will print the entire text buffer.

H 100,500

This command will print lines numbered 100 through 500 inclusive.

H.

This command will print the single line pointed to by the current line pointer, ".".

H

This command will print the 15 lines (Model II and LDOS 6.x print 23 lines) starting with the current line.

<I>NSERT

This command is used to invoke the <I>nsert mode so lines can be input into the text buffer. <I>nsert is somewhat similar to the "AUTO" command in BASIC. <I>nsert's syntax is:

```
I {line#{,inc}}

line#       is the number of the line that the insert
            should follow.

inc         changes the current increment to "inc".

Note: use <BREAK> or <SHIFT-CLEAR> to exit
```

The Insert command is used to insert or add text lines into the text buffer. All lines of source text are entered with the use of the <I>nsert command. After using the <I>nsert command to specify where you wish to place new lines, the editor will generate the designated line number and allow the inserting of that numbered text line. After entering the first text line the editor will generate the next line number higher, as specified by your increment selection. Incremental line numbers will continue to be generated as long as there is room between lines or room left in the text buffer.

If a desired increment is not specified, the last specified increment is assumed. Period, ".", may be used for "line#" to indicate the current line or if "line#" is omitted, the current line will be assumed.

The <BREAK> key will allow you to leave the insert mode at any time. The <CLEAR> key also performs a functional BREAK. If you have entered the <BREAK> before depressing <ENTER> to complete the input of a line, that line will not get entered into the text buffer.

Examples of the <I>nsert command:

```
I 300,5
```

This command will begin the text insertion to follow line numbered 300 and also change the increment to 5.

```
IB
```

This command will append new text to the end of the text buffer. It is identical to performing a "Pb" followed by an "I".

<K>ILL

This command can be used to erase a file from a disk. It will function identically to the DOS KILL (or REMOVE) command. Its syntax is:

```
K filespec

filespec    is the filespec of the file to be erased.

Note: The file extension currently in effect for "source"
      files will be used as a default extension.
```

This command is used to delete a file that is not needed. Coupled with use of the QUERY command, file maintenance can be implemented from within the Editor Assembler environment. This is especially useful when a <W>rite command results in a **DISK FULL** DOS error and you have to find a diskette with sufficient free space.

In order to guard against inadvertant use of the <K>ill command, a filespec must be entered. If no extension is entered, the extension currently in effect for source files (usually "ASM" unless over-ridden by LC or EXT= parameters) will be assumed. If you enter the <K>ill command without a filespec, the message:

```
Bad parameter(s)
```

will be displayed.

Note: The <K>ill command is not available on Model II versions of EDAS. Therefore, one must use the <Q>uery KILL DOS command on the Model II.

Examples of the <K>ill command:

```
K OLDPROG/ASM:2
```

This command will erase the file, OLDPROG/ASM, from drive 2.

```
K TEST:0
```

This <K>ill command will erase "TEST/ASM" from drive 0.

<L>OAD

This command is used to load a source file into the text buffer. Its syntax is:

```
L {filespec}

filespec    is the filespec of the file to be loaded.
```

The <L>oad command will read the file denoted by the "filespec" into the text buffer. The text file will be concatenated to any text already in the text buffer. The file specification is composed of a FILENAME, optional EXTension, optional PASSWORD, and optional DRIVE reference as in:

```
FILENAME/EXT.PASSWORD:D
```

If you do not enter the "filespec", EDAS will prompt you for the filespec. If you omit the file extension (EXT), a default extension of "ASM" will be used thus saving keyboard input and at the same time providing for a standard file naming convention. If the "LC" parameter was specified in the EDAS command line, then "CCC" will be used for the default. The EDAS parameter "EXT=ext" can be used to override the assigned default extension to that of "ext" (see the chapter on RUNNING EDAS).

The <L>oad command will automatically handle a source file that is line-numbered and headered (EDAS Version III format), line-numbered and un-headered (EDTASM Series I format), or un-numbered and un-headered (EDAS format, text editor prepared files, or certain M-80 files). Model II source files created with EDAS 4.0 must be converted using the CONV40 utility. If the file being read is not line-numbered, EDAS will automatically number it as it loads. A line number counter is kept internally that advances by the current increment for each un-numbered line read. Thus, concatenation of source text via multiple loads of un-numbered source files will produce a sequentially numbered in-memory text. The line number counter is reset to its initial starting value only by a warm-start or depression of the <CLEAR> command function.

A line-numbered file is interpreted as one in which the first five characters of a line have the high-order bit (bit 7) set. The 5-character line number is also followed by a terminating character (usually a space but could be a tab with bit 7 set). A headered file is interpreted as one in which the first character of the file is an X'D3'.

"ASCII" files prepared by a word processor program (i.e. SCRIPSIT) are loadable by EDAS; however, they must be pure ASCII and must have line lengths not exceeding 128. The only requirement is that there must be an end-of-file character as the last character of the text (which would follow a carriage

return). The end-of-character can be either an X'1A' or a null, X'00'. EDAS can only convert lower case to upper case during <I>nput or <E>diting so if you use an external word processor program, keep the Z-80 code in upper case.

Examples of <L>oad commands:

L myprog

This command will search for a file named "MYPROG/ASM" (assuming a default extension of "ASM") and load it into the text buffer.

L theprog:1

This command will load the file named "THEPROG/ASM" from drive 1 into the text buffer.

Dt,b
L newprog:2

This sequence of commands will first clear the text buffer then load the file named "NEWPROG/ASM" from drive 2.

<M>OVE

This command is used to <M>ove a line or block of lines from one text buffer location to another. Its syntax is:

```
M line1, line2, line3

line1       is the line number of the first line to move.

line2 :     is the line number of the last line to move.

line3       is the number of the line that the block
            should follow after the move.
```

This command is used to move a block of lines from one location in the text buffer to another. A large quantity of text lines can be moved to a different position in one easy operation. In the command syntax, "line1" and "line2" are the beginning and ending line numbers of the text block to be moved. "Line1" and "line2" are permitted to reference the same line number if only one line is to be moved. "Line3" is the line number of the line that the text block will follow after the move. The line number references must be offset by commas ",". Your line number parameters must specify existing lines in the text buffer. If any of the entered line numbers are non-existant, the message,

No such line

will be displayed.

"Line3" is not permitted to equal "line1" or "line2" as that would represent an illogical move operation. "Line3" is not permitted to be a line interior to the range "line1" through "line2" as that would also be an illogical operation. The message,

Bad parameter(s)

will be issued if your input violates any of these conditions.

The block of text to be moved is stored temporarily in the spare text region. If this region is not large enough to store the block, the message,

Text buffer full

will be issued. Try moving the block in smaller segments.

Upon completion of the move, all lines in the text buffer will be renumbered starting from 100 and incremented according to the line increment

currently in effect. Renumbering is absolutely essential to perform proper operation of Editor Assembler commands and so it is done automatically.

Examples of <M>ove commands:

M 500,900,1510

You desire to move the block of text starting at line 500 and ending at line 900 to follow line 1510. This command will perform the desired operation.

RE<N>UMBER

This command is used to re<N>umber the lines of text in the text buffer. Its syntax is:

```
N {line{,inc}}

line        is the new first line number.

inc         is the new increment.
```

The <N> command is used to renumber the lines in the text buffer. The first line in the buffer is assigned the number specified as "line". If "line" is not specified, it defaults to ØØ1ØØ. The remaining lines in the buffer are renumbered according to the increment "inc" or the previous increment in a re<N>umber, <R>eplace, or <I>nsert command if the increment was not specified. The current line pointer, ".", points to the same line as it did before the re<N>umber command was used, but the actual number of this line may be changed.

Examples of line re<N>umbering:

N

This command will renumber the text to start with line number 1ØØ. The previous increment in effect will be used.

N5

This re<N>umber command will renumber the text to start with line number 5. It also uses the previous increment.

N1Ø,5

This command will renumber the text to start with line number 1Ø. It changes the line increment to a value of 5.

<P>RINT

The <P>rint command is used to display a line or block of lines to the video display. Its syntax is:

P {line1{,line2}}	
line1	is the number of the first line to display.
line2	is the number of the last line to display.

The <P>rint command will display a line or a group of lines on the monitor screen. The current line pointer, ".", is updated to point to the last line displayed.

If "line1" is entered without entering "line2", then only "line1" will be displayed. If neither "line1" nor "line2" are entered, then the current line plus 14 additional lines (total of 15) will be displayed (23 total lines will be displayed on the Model II).

Examples of <P>rinting lines:

P #,* or P t,b

This command will display all lines in the text buffer. You may use the <PAUSE> function to temporarily halt the display from scrolling.

P 100,500

This command displays lines 100 through 500 inclusive.

P .

This command will display the the line pointed to by the current line pointer. Only a single line will be displayed.

P

This command displays 15 lines (23 on the Model II) starting with the current line. The <P>rint command operates in a screen scroll mode.

<Q>UERY

On the Model I or III, this command can be used to obtain a directory of files stored on a disk. Under LDOS 6.x or on the Model II, <Q>uery is used to execute a DOS command. Its syntax is:

Model I/III	
Q{d{/ext}}	
d	is the drive (Ø-7) for which a directory display is desired.
/ext	is an optional "part-spec" file extension used to display only files matching the "ext".

LDOS 6.x or Model II	
Q DOS-command	
DOS-command can be any DOS command except COPY or BACKUP	

With Model I or III, this command is used to display a directory from the designated drive. If a drive number is not entered, drive Ø will be assumed. The "part-spec" optional entry can be useful to isolate the directory display to select only those files matching a particular class. For example, if you only want to display the names of "/ASM" files, the part-spec extension should be used.

Under LDOS 6.x or on the Model II, <Q>uery is used to interface with the DOS while in the evironment of the Editor Assembler. Any DOS command can be accessed. It is recommended that you not attempt to access the "COPY" or "BACKUP" commands due to the possibility of overwriting the Editor Assembler.

IMPORTANT: NEVER DEPRESS <BREAK> ON THE MODEL II DURING A DOS COMMAND EXECUTION. TO BREAK ANY DOS COMMAND, USE THE <ESCAPE> KEY.

Examples of <Q>uery commands:

Q DIR

This LDOS 6.x or Model II <Q>uery command will list the diskette directory to the display device.

Q1/CCC

This Model I or Model III <Q>uery command will display the names of all LC source files stored on drive 1.

<R>EPLACE

This command can be used to replace a specified text line and automatically enter <I>nsert mode. Its syntax is:

R {line{,inc}}
line is the number of the line to replace.
inc is the new increment to be used.

The <R>eplace command only replaces the one line specified and then enters <I>nsert mode. If "line" is omitted, then the current line is assumed. If "line" exists, it is deleted and then <I>nsert mode is entered starting with that line number. If "line" doesn't exist, <I>nsert mode is entered just as if the <I>nsert command were invoked. If "inc" is not specified, the last increment specified by an <I>nsert, <R>eplace, or re<N>umber command is used. The current line pointer, ".", is always updated to the new current line.

If during subsequent INPUT of lines, the error message:

No more room

is issued, it means that a line numbered "current" + "inc" already exists. It is suggested that you renumber the lines and continue your insertion after ascertaining the new line number assigned to the "current" line.

Examples of <R>eplace commands:

R

This command will replace the current line.

R 100,10

This <R>eplace command will start replacing lines beginning at line numbered 100 and enter <I>nsert mode with an increment of 10.

R 100

This command will start replacing at line numbered 100 using the last specified increment.

<S>WITCH CASE CONVERSION MODE

This command is used to toggle the "case conversion mode" of EDAS. It will either permit the acceptance of both upper case and lower case, or invoke the automatic conversion of lower case to upper case where required. Its syntax is:

```
S

There are no parameters or options.
```

Command <S>witch will toggle the switch-case conversion of lower case to upper case. If your computer supports the display of lower case, this feature will be of great benefit. Two modes are available:

1. Lower case accepted: This mode permits entry of either lower case or upper case. Your input is preserved in whatever case it is entered. EDAS is suitable as a text editor in this mode. This is the mode used when entering LC C-language source text.

2. Lower case converted: This mode permits entry in either upper case or lower case. All lines are converted to upper case during <I>nput mode or when exiting the <E>dit mode. This mode should be used to input assembler source text. While in the lower case converted mode, the following conversion behavior is exhibited:

   Character strings within single quotes are kept in lower case if entered in lower case. This will ensure that your string declarations are kept intact.

   Characters entered following a semi-colon are kept in lower case if entered in lower case. This permits the entry of comments in lower case which makes your source text much more "readable".

On entry to EDAS, the "lower case converted" mode is activated. Each entry of an "S" command will switch (toggle) the case mode and an appropriate message will be displayed.

Lower case permitted - for full lower case
Lower case converted - for upper case conversion

Since the <I>nsert command mode converts to upper case, the <F>ind and <C>hange commands utilize the <I>nsert input and will also convert to upper case. You can <F> or <C> lower case by using the case switch toggled to "lower case permitted".

<T>YPE

This command can be used to print a line or block of lines on a line printer. In contrast to the <H>ard copy command, <T>ype will omit the line numbers. Its syntax is:

T {line1{,line2}}	
line1	is the number of the first line to print.
line2	is the number of the last line to print.

The <T>ype command prints a line or block of lines onto the Line Printer. The current line pointer, ".", is updated to point to the last line printed. This command is much like the <H>ard copy command, except line numbers are not printed. Only the source text is printed. If a properly paged display is desired, it is suggested that you set your paper to begin printing at the sixth line from the top of the page (for additional information on paging, see the <1> command).

Examples of <T>ype commands:

For examples of the <T>ype command, see the <H>ard copy command. The two commands function identically except that <T>ype omits the line numbers during the printing.

MEMORY <U>SAGE

This command is used to display certain statistics concerning the memory usage of your source text buffer. Its syntax is:

```
U

There are no parameters or options.
```

This command will display the number of bytes of text buffer in use, the number of bytes spare and the first address available for assembly to memory (note that if macros are being used, the macro storage area extends from the address shown as the first address available for assembly and you will have to experimentally choose a higher address for an "in-memory" assembly).

This command is useful to ascertain requirements for storing the text buffer to disk. Note that a disk file, which is written in ASCII (un-numbered), will contain two (2) bytes less per text line. The 2 bytes represent the line number used in the storage format of text in memory versus text in an un-numbered ASCII file.

It also is useful when assembling into memory. Since the Assembler will not permit you to overwrite it or the text buffer, you will have to "ORG" your program in the free text buffer area. The first available address is output by this command (remember the note on macro storage).

An example of <U>sage output is:

```
30622 bytes spare
00000 bytes in use
8863H is the first free address
```

<V>IEW

This command is used to list (display) a file on the video display device. Its syntax is:

```
V {filespec}

filespec    is the filespec of the file to be displayed.
```

This command can be used to display any file without actually loading the file into the text buffer. No attempt is made to convert non-ASCII characters prior to displaying. Therefore, if the file is not an ASCII file, strange characters may be displayed. Use the <V>iew command primarily to display source files.

The output may be temporarily stopped by depressing the <PAUSE> key or may be interrupted and cancelled by depressing the <BREAK> key.

If you do not enter the filespec with the command line, it will be prompted for with the query:

filespec?

If you do not enter a file extension with the file specification, a default extension of "ASM" will be used unless the "LC" parameter was specified when entering EDAS. "LC" redefines the default specification to "CCC". Note that the default extension could also have been changed via the "EXT=ext" parameter.

<W>RITE

The <W>rite command is used to save the contents of the text buffer into a disk file. Its syntax is:

W{+}{#}{$}{!hh} {filespec}	
filespec	is the filespec to be written.
+	is an optional switch to write a source file created with a header record.
#	is an optional switch to write a source file with line numbers.
$	is an optional switch to write a source file with line numbers terminated by X'89'.
!hh	is an optional switch to specify a end-of-file terminating byte of X'hh' other than X'1A'. Use "!!" to suppress the E-O-F byte.

This command will write the text buffer to the file denoted by filespec. If no filespec is entered, you will be prompted for it in a manner identical to the <L>oad command. If you omit the file extension (EXT), a default extension of "ASM" will be used thus saving keyboard input and at the same time providing for a standard file naming convention. Remember, if you had specified "LC" or "EXT=ext" when you entered EDAS, the default source extension will be "CCC" or "ext" respectively.

The switches are used for compatibility in writing source files for use with other editors such as the M-80 editor, EDIT80, earlier versions of EDAS (3.4 and 3.5), and EDTASM. If more than one switch is used, the order is irrelevant. Use of the switch "+" will enable creating a file with a file header record (X'D3' followed by a 6-character filename).

If the source file is to contain line numbers, then the "#" switch should be used. This will write line numbers as five ASCII digits with the high order bit (bit-7) set. The line number is terminated with a space character (X'20'). The switch "$" generates a line numbered file the same as the "#" switch; however, the terminating character is written as a tab with bit-7 set (X'89'). Some versions of FORTRAN require the source file to be in this manner; thus, EDAS could be used to prepare source files for FORTRAN.

Finally, the "!hh" switch can be used to specify an end-of-file byte to be other than the standard X'1A' normally used by EDAS. For instance, specifying "!00" will change the E-O-F byte to X'00', the value used by

SCRIPSIT. If instead of the two-character hexadecimal value, you enter a second exclamation point as in "!!", then no E-O-F byte will be written. Observe caution as EDAS can only properly load a file if the E-O-F byte is an X'1A' or an X'ØØ'.

If the file denoted by "filespec" is non-existant, a file will be created and the message,

New File

will be issued. If the file denoted by "filespec" is an existing file, it will be replaced by the write operation and the message,

Replaced

will be issued. YOU WILL NOT BE GIVEN AN OPPORTUNITY TO CANCEL A WRITE REQUEST ON AN EXISTING FILE. Know what you are doing.

Examples of <W>rite commands:

W parmdir1:3

This command will write the current contents of the text buffer to the file, PARMDIR1/ASM:3

W !ØØ doparm/jcl:Ø

This <W>rite command will save the text buffer in the file, DOPARM/JCL:Ø. An E-O-F byte of X'ØØ' would be used instead of X'1A'. Thus, EDAS was used to edit a Job Control Language file.

E<X>TEND

This command can be used to increase the area of the text buffer by eliminating the assembler. Its syntax is:

```
X

There are no parameters or options.
```

This command can be used to extend the text buffer area by moving the text over the Assembler portion of EDAS in memory. Approximately 8000 bytes are gained by this extend operation. It is useful if you are editing a large body of text or are dealing with a large assembly language source program. Since the capability of direct assembly from disk files is a function of the EDAS Editor Assembler, editing can be performed without the Assembler module of the program in memory. You, of course, will have to exit and reload the Editor Assembler for further assembling.

Another reason for the use of e<X>tend, is to handle those EDAS 3.5 files that now exceed the maximum text buffer size of EDAS version IV. It is suggested that you keep your source files in smaller modules. The *GET capability provides great power in handling multiple source files in an assembly stream. You will thus find that a program made up of smaller modules of code is perhaps easier to maintain and just as fast to assemble.

Following the entry of the <X> command, the prompt:

Are you sure?

will be displayed. This is provided as a safeguard in case you inadvertantly enter the <X> command. You must respond <Y> in order to complete the extension. Entry of any other character will abort the extend operation. A response to the query with a <Y> will move the current contents of the text buffer and reset all pointers to their proper value. Once the e<X>tend command is invoked, both it and the <A>ssemble command will be made inoperative.

<1> (ONE)

This command can be used to display or alter the current page formatting parameters of EDAS. It is not supported under LDOS 6.x or Model II (use Q FORMS). Its syntax is:

1{n1{,n2}}
n1 is the number of lines to print per page.
n2 is the page length in lines.

This command can be used to alter the two paging parameters used by EDAS. One of these parameters specifies how many lines to print on a page before issuing a form feed. The other parameter is specified in the printer Device Control Block (DCB) and represents the maximum printing lines on a page. EDAS initializes with "n1" set to 56 (57 on a Model III since a Model III starts counting from 1). Thus, 56 lines will be printed before sending a page eject. The value of the page length stored in the *PR DCB (X'4028' Model I and III) is used for the "n2" value. Either value can be changed with this command. If no parameter is entered, then the current values will be displayed.

Examples of the <1> command:

1 46 51

This command will set the maximum page length to 51. The number of printed lines until a form feed is generated will be set to 46.

1

This command will display the current values for lines-to-print and lines-per-page. The display will look like:

```
00056 00066      (Model I and Model II)
00057 00067      (Model III)
```

MESSAGE TO JOB LOG "."

The dot "." command can only be used with LDOS, to post a time-stamped message to an active job log. There will be no visual indication of the event. Its primary utility will be with Job Control applications of EDAS. An example of a message post would be:

```
. Starting assembly of PARMDIR
```

SCROLL UP <UP-ARROW>

The "SCROLL UP" command displays the line preceding the current line and updates the current line pointer, ".", to point to the line displayed. If the current line is the first line in the text buffer, it is displayed and period "." remains unchanged. "SCROLL UP" is an immediate command and must be the first character of a command line in order to be interpreted.

SCROLL DOWN <DOWN-ARROW>

The "SCROLL DOWN" command displays the line following the current line and updates the current line pointer, ".", to point to the line displayed. If the current line is the last line in the text buffer, the last line is displayed and period "." remains unchanged. "SCROLL DOWN" is an immediate command and must be the first character of a command line to be interpreted.

CLEAR SCREEN <SHIFT-CLEAR (Model I/III)> <F1 (Model II)>

The <CLEAR> key is used to perform a functional clear screen and display of the initial entry message. The "CLEAR" function also performs a <BREAK> operation but cannot be used to interrupt output. This function is identical to a warm-start of EDAS and will reset automatic line numbering to its intital value of 10.

On the Model I and Model III, the <SHIFT-CLEAR> key performs the "clear" function. The <F1> key is used on the Model II. Consult your DOS manual for the appropriate key under LDOS 6.x.

PAUSE <SHIFT-@ (Model I/III)> <HOLD (Model II)>

The <PAUSE> key is used to pause the computer during a display, during any assembly, or Editor Assembler printing. When a pause is sensed, depression of any key except <PAUSE>, <SHIFT>, or <CONTROL> will continue the operation paused. It is only necessary to momentarily depress the key as a pause function will be held pending as soon as the key is pressed. On the Model I and Model III, the <SHIFT-@> key is used as a "pause". The <HOLD> key is used for this purpose on the Model II.

BREAK

The <BREAK> key is used to terminate the <I>nsert mode. It is also used to abort an assembly in effect. It will also abort any disk I/O operation or display listing. A detected <BREAK> will return EDAS to the command ready prompt, ">".

PAGE FORWARD <SHIFT RT-ARROW (Model I/III)> <F2 (Model II)>

The <SHIFT-RT-ARROW> key on the Model I and Model III is used to advance the display by 15 lines. The <F2> key is used on the Model II to advance the display by 23 lines. This command is similar to the <P>rint command except that the display screen is cleared prior to displaying the 15/23 lines of source text.

USER PATCH SPACE - ZCMD

A 50-byte patch space is available for your use. A vector pointing to this space is located at X'5809' (Model I/III), X'3609' under LDOS 6.x, or X'3709' (Model II). If you place a routine in this space, it can be executed by entering a <Z> at command ready. The space currently has a RET instruction as the first byte which is used to return from the <Z> command.

X R E F

The MISOSYS XREF utility is used to generate a cross reference listing of symbols used in your source code. Its syntax is:

XREF filespec/REF {(LEN=val,PAGE=val,LINES=val,EQU,LIMIT)}

filespec	is the specification of the reference data file generated by the -XR switch of EDAS. If the file extension is omitted, "REF" is used.
LEN	is the length of your print line (the default value is 80).
PAGE	is the maximum number of lines per page (the default is 66 for Mod I & II, 67 for Mod III).
LINES	is the number of lines to print on a page (the default is 56 for Mod I & II, 57 for Mod III).
EQU	is used to generate a file of EQUates instead of the cross reference listing.
LIMIT	is used to limit the file of EQUates to those symbols containing a special character.

Note: the format of "value" is PARM=ddd or PARM=X'hhhh'.

PAGE and LINES are not supported under LDOS 6.x or Model II

There are no parameter abbreviations.

The XREF/CMD utility generates a symbolic cross-reference listing which includes a sorted list of all defined labels, the file of origin of the definition, the line number of the definition, the value of the definition, and the line numbers of all statements referencing the label. If "*GET" or "*SEARCH" files are used in the assembly process, XREF will even identify the filename of the file containing the references. XREF will not identify unresolved labels. Therefore, make sure that either all labels are resolved during the assembly that generates the XREF data file, or you do not need the line numbers of those unresolved references appearing in the cross reference listing.

XREF can also be used to generate an assembler source file of EQUates of all symbols used in the program being assembled or a subset of all symbols used. The LIMIT parameter is used to limit the EQUates to only those symbols having at least one special character in the symbol name.

XREF uses, as input, the reference data file which is optionally generated by the -XR switch during the LISTING pass of EDAS (phase 2). XREF cannot function without this data file. You need not enter the file extension, /REF, as it will be assumed if omitted.

The XREF command line parameters enclosed in parentheses are entirely optional. The may be used as follows:

LEN

This parameter controls the printed line length during the XREF listing. If omitted, a value of 80 is assumed to deal with 80-column line printers. If you are using a wide-carriage printer (typically 132 columns), then XREF can use the entire print line by specifying the parameter as:

```
XREF (LEN=132)
```

PAGE

This parameter controls the page size. A value of 66 lines per page (67 on the Model III due to its line counter starting from 1 instead of 0) is used. If your paper is shorter or longer, you can respecify the page length from the command line. For instance:

```
XREF filespec (PAGE=51,LINES=41)
```

will set the page length to 51 lines per page and initialize to print 41 lines.

LINES

This parameter controls the quantity of lines printed on a page before a page eject (form feed) is generated. If omitted, a value of 56 printed lines is used. You can respecify the quantity of lines you want printed by a command similar to that shown for the PAGE parameter.

EQU

This parameter controls the generation of the EQUate file. If this parameter is entered, then the cross reference listing is suppressed and a source file of symbols equated to their value is generated. The filespec used to write the EQUate file will be constructed using the filename and drive specification of the "/REF" file. A file extension of "/EQU" will be used. If this parameter is entered, then LEN, PAGE, and LINES will be ignored.

Symbols defined by the "DEFL" pseudo-OP will be maintained as DEFL's in the EQUate file. The file will be created without a header and without line numbers - it will be a standard EDAS Version IV file.

LIMIT

This parameter controls what symbols are written to the EQUate file. If entered in addition to the "EQU" parameter, then the EQUate file will be limited to those symbols that contain at least one special character (a character other than A-Z, Ø-9).

Cross-Reference Listing

The listing requires two passes through the data file. This is done to conserve memory space so that listings for extremely large programs can be processed. If you are generating the cross reference listing, three informative messages will be displayed prior to generating the printer output. "Building symbols declared" will be displayed during the first pass through the data file as XREF creates a table of information pertinent to all symbols declared. After this table is completed, the message, "Sorting symbol table" will be displayed. The operation being performed is self evident. A second pass through the REF data file will be made while the message, "Building symbols referenced" is displayed. This pass is used to create a second table of information pertinent to all references to symbols.

The listing will contain a heading on each page. This heading is composed of the system DATE and TIME, the TITLE extracted from the source code if a TITLE pseudo-op was used in the assembly process, and a page number. The heading line requires a minimum of 74 columns. Thus, if you specify a LEN parameter of less than 74, the heading will either wrap around on your printer or be truncated - depending on how your printer handles longer lines. The reference columns will include:

Origin

The filename of the file containing the declaration of the symbol. If the symbol was declared by a statement located in memory, the ORIGIN will be listed as "$MAIN". Otherwise, the ORIGIN will list the filename of the "*GET filespec" or "*SEARCH library".

Symbolic Label

This column contains the symbol name of the declaration. If the symbol was defined by a "DEFL" pseudo-OP, a plus sign, "+", will precede the symbol name to denote this fact. - references will only be printed against one of the label definitions; however, all declarations will be shown. If the symbol name was actually the name of a MACRO, it will be prefixed by a pound sign, "#" and the "value" field will be irrelevant. The symbolic labels are sorted in ascending alphabetical order.

Value

This column contains the value of the symbol as determined during the assembly process. If the symbol shows a DEFL definition, the value will be the first defined value. If a MACRO name is indicated, the value shown is actually the storage location of the MACRO prototype and model - it will serve no useful purpose.

Line#

This column contains the line number of the source line declaring or defining the symbol. The symbol is defined where the symbolic name is used in the label field of a source statement.

Usage

This column contains the filename of the file containing a reference to the label. If the label is referenced from a statement resident in memory, then the filename will be listed as $MAIN. Otherwise it will be the filename field of the *GET filespec pseudo-OP fetching the file or the library filespec if a *SEARCH was involved.

Line# of References

All references to the label will be listed in this field. It will contain the line number of the source statement containing the reference. All of the references listed on a print line will be contained in the file identified under the usage column. Whenever the Usage file changes, it will cause a new line to be generated in the listing.

Statistics

At the conclusion of the cross reference listing, two additional items of information are listed. The quantity of symbols declared is listed along with the quantity of references associated with those declarations.

T T D

The MISOSYS TTD utility is used for transferring to disk, a source cassette file that was created with the Radio Shack EDTASM, Microsoft EDTASM+, or other compatible editor assembler. TTD is not supported under LDOS 6.x or on the Model II.

To execute the TTD utility, at your DOS ready, simply use the syntax:

```
TTD (:d)
```

TTD is used to transfer a source cassette file to disk. The filespec will be constructed using the filename found on the cassette tape file and the file extension "/ASM". If the optional drive specification, ":d" (where "d" is the drive number of the drive receiving the disk file), is entered with the TTD command line, it will be used in the construction of the file specification.

TTD will prompt you to ready the cassette via the message:

Ready cassette and <ENTER> -> for a Model I
Ready cassette and enter <H,L> -> for a Model III

The <H,L> entry for Model III users will select either High speed cassette operation (15ØØ baud) or Low speed cassette operation (5ØØ baud). Respond to the prompt by depressing the <ENTER> key if you are a Model I user, or the correct baud rate character if you are a Model III user.

The cassette source file will be transferred to disk. TTD will then return to DOS.

Error Messages

GENERAL

EDAS Version IV recognizes three types of errors. These are:

Command	This is an EDAS command syntax error. The error message is displayed and control is returned to command mode.
DOS	This is an operating system disk I/O error. The error message is displayed and control is returned to command mode.
Assembler	These errors may occur while executing an Assemble command. There are three types: terminal, fatal, and warning.

DOS disk I/O errors can also be received during an assembly. When a disk I/O error occurs, the assembly will be aborted and control will be returned to EDAS command ready.

Three different types of assembler errors can occur. The types relate to the severity of the error. These types are:

Terminal	Assembly is terminated and control is returned to command mode.
Fatal	Processing of the line containing the error is immediately stopped and no object code is generated for that line. Assembly proceeds with the next statement.
Warning	The error message is displayed and assembly of the line containing the warning continues. The resulting object code may not be what the programmer intended.

Following is a list of all error messages and an explanation of each.

COMMAND ERRORS

1.> Buffer full

There is no more room in the text buffer for adding text.

2.> Bad parameter(s)

Any command line not entered according to the syntax appropriate for that command will generate this error message. Also, if you attempt to load a file that is not a valid source code file, this message may be displayed. The <K>ill command requires entry of a filespec, which if omitted, will also display this error message.

3.> Illegal command

The first character of the command line entered does not specify a valid Editor Assembler command.

4.> Line number too large

Renumbering with the specified starting line number and increment would cause line(s) to be assigned numbers greater than 65529. The renumbering is not performed. This message would also be displayed if you attempted to INSERT a line with a line number exceeding 65529.

5.> No room between lines

The next line number to be generated by INSERT or REPLACE would be greater than or equal to the line number of the next line of text in the edit buffer. The increment must be decreased or the lines in the buffer renumbered.

6.> No such line

A line specified by a command does not exist. The command is not performed.

7.> No text in buffer

A command requiring text in the buffer was issued when the text buffer was empty. The commands <L>oad, <I>nsert, <Q>uery, <S>witch, <B>ranch, <U>sage, <V>iew, e<X>tend, <K>ill, Dot <.>, <Z>, and ONE <1> can be executed when the text buffer is empty. All other commands require at least one line of text to be in the buffer.

8.> String not found

The string being searched for by the <F>ind command could not be found between the current line and the end of the text buffer. This message will also be displayed at the completion of a global change command.

DOS ERRORS

The standard DOS error messages will be displayed if the DOS returns an error code after return from any disk operation. Consult your DOS operating manual for explanations of those errors. During most error handling, the abbreviated form of the error message will be displayed. If an I/O error is detected during an assembly, the long form of the error message will be displayed. This provides an observance as to which file was affected by the I/O error.

Any attempt to load or *GET a file that has a line longer than 128 characters will result in "Load file format error".

TERMINAL ERRORS

1.> Memory overlay aborted

During an assembly to memory, a block of code was assembled that would load into a memory region other than the spare text buffer area. Your program will not be permitted to load to an address below the end of the text buffer or above the symbol table. Use the Usage command to locate the first available memory address. If you are using MACROs, the first available memory address is indeterminate as the MACRO processor uses the memory area immediately following the text buffer for a MACRO model and string buffer storage area.

2.> Symbol table overflow

There is not enough memory for the assembler to generate your program's symbol table. You have three options:

1.> Remove comment lines and/or comments following Z-80 code operands. This may free up enough space to perform the assembly.

2.> Divide your program into two or more modules and assemble them using the *GET filespec directive.

3.> Extend the text buffer area, expand your source, then assemble it using the *GET filespec directive.

3.> *GET or *SEARCH error

A "*GET filespec" or "*SEARCH library" assembler directive was found in a library member. A searched library cannot have "*GETs" or nested "*SEARCHes".

4.> Member definition error: filespec(member)

This is a result of a fetched *SEARCH member not resolving the symbol reference invoking its fetch.

FATAL ERRORS

1.> Bad label

The character string found in the label field of the source statement does not match the criteria specified under ASSEMBLY LANGUAGE INFO - LABELS.

2.> Expression error

The operand field contains an ill-formed expression.

3.> Illegal addressing mode

The operand field does not specify an addressing mode which is legal with the specified OPCODE.

4.> Illegal opcode

The character string found in the opcode field of the source statement is not a recognized instruction mnemonic, assembler pseudo-op, or MACRO name.

5.> Missing information

Information vital to the correct assembly of the source line was not provided. The OPCODE is missing or the operands are not completely specified.

6.> Too many nested *GETs

*GET filespec nesting exceeds the number of levels supported. The *GET will be ignored.

7.> Unclosed conditional

The "END" statement or end of source was reached and an open "IF" conditional block was still pending. Your program is missing the closing "ENDIF".

8.> ENDIF without IF

An "ENDIF" pseudo-op was detected without a corresponding conditional "IF" or "IFxx" in effect. The "ENDIF" will be ignored.

9.> ELSE without IF

An "ELSE" statement was detected without a preceding "IF" conditional segment.

10.> Filespec required

A *GET or *SEARCH directive was detected but the statement did not contain the required file specification. The *GET or *SEARCH will be ignored.

11.> Bad parameter(s)

When output preceding a MACRO definition, it implies an error in the parameters of a MACRO.

12.> Nested MACRO ignored

A macro definition statement was nested in the model of another macro.

13.> Missing MACRO name

The name field of the macro definition statement did not contain the macro name. The macro will not be defined.

14.> ENDM without MACRO

An ENDM pseudo-OP was detected while not in a macro definition phase. It will be ignored.

15.> Too many parameters

In a macro call, the number of parameters passed exceeded the number defined for the macro. The macro call will not be expanded.

16.> Too many nested MACROs

The number of pending nested macro calls exceeds the current nest level supported. The macro call will not be expanded.

17.> MACRO forward reference

A macro call was detected prior to the definition of the macro. The macro call will not be expanded since gross phase errors would result.

18.> Multiply defined MACRO

A macro definition statement was detected for a macro already defined. The subsequent definition will be ignored.

WARNINGS

1.> Branch out of range

The destination of a relative jump instruction (JR or DJNZ) is not within the proper range for that instruction. The instruction is assembled as a branch to itself by forcing the offset to hex X'FE'.

2.> Field overflow

A number or expression result specified in the operand field is too large for the specified instruction operand. The result is truncated to the

largest allowable number of bits. This error would also be output during a global change if a resultant line would exceed 128 characters.

3.> Multiply defined symbol

The operand field contains a reference to the symbol which has been defined in another line. The first definition of the symbol is used to assemble the line.

4.> Multiple definition

The source line is attempting to illegally redefine a symbol. The original definition of the symbol is retained. Symbols may only be redefined by the DEFL pseudo-OP and only if they were originally defined by DEFL.

5.> No END statement

The program END statement is missing. Note that if your program is missing the "END" statement, EDAS cannot detect an unclosed conditional. Also, be aware that if your program has a FALSE unclosed conditional, then the "END" statement will NOT be detectable - even if present.

6.> Undefined symbol

The operand field contains a reference to a symbol which has not been defined. A value of zero is used for the undefined symbol.

OBJECT FILE FORMAT

The disk file object code format consists of a header record, an optional comment record, one or more load block records, and a transfer address record. The specific formats of these records are as follows:

Header Record

The file header record consists of the hex byte X'Ø5' (record type) which indicates the header field of an object file. It is followed by the header length byte which indicates the length of the header data following. The length of the header data is constant in EDAS and is six bytes. The data is constructed as the first six bytes of the object code file name field and is filled out with spaces if the file name is less than six characters.

Comment Record

This record is optional. It is generated by the "COM" pseudo-OP. It consists of a record type byte of X'1F' followed by a length byte which is the length of the comment. The comment data, itself, follows.

Load Block Record

The load block record starts with a record type code of X'Ø1' which indicates it is a load block. A 1-byte length is next. This indicates the length of the object code data plus the 2-byte block load address. The length is encoded as a modulo 256 value (object code length of 253 = X'FF', object code length of 254 = X'ØØ', object code length of 256 will show as X'Ø2').

The block length byte is followed by the 2-byte block load address which is the address that will be loaded with the first byte of the block.

Finally the object code block immediately follows for as many bytes as two less than the block length.

Transfer Address Record

The Transfer address record (TRAADR) starts with a record type of X'Ø2'. An X'Ø2' is written to indicate the length of the entry point address. This is then followed by the 2-byte entry point or transfer address generated from the label or constant in the operand field of the assembler source END statement. As is the case with all 16-bit data values, the TRAADR data has the low-order byte of the address followed by the high-order byte.

SOURCE FILE FORMAT

The source code file format used by EDAS has no header nor line numbers. Headers and numbers are entirely optional and can be generated with appropriate switches in the <W>rite command. The formats are as follows:

Header Record

A header record as described under "Object file format" is optionally used for source files with the exception that the first byte is a hex X'D3' (X'53' - with bit 7 set) to identify the file as source, immediately followed by a 6-character name (the name length byte is omitted). Files written with "W+" contain this header.

Text Lines

Text lines are written in ASCII each composed of an optional 5-character line number (bit 7 is set), a space, the text line, ending with an <ENTER> (X'ØD'). Files written with the "W#" command incorporate both the 5-character line number and following space.

End-of-File Mark

The file end is indicated by an end-of-file mark of X'1A' which would be in the first character position of a text line (or 1st byte of the line number if line numbered files are used).

REFERENCE DATA FILE FORMAT

The reference data file is a compressed collection of data corresponding to each symbol definition and reference. The file contains a title record, and definition/ reference records. The format of these records is as follows:

Title Record

The title record is always present even though the assembler source file stream may or may not have supplied a TITLE pseudo-OP. The title record is 28 characters long. If the source files did not contain a TITLE pseudo-OP, the record will be filled with spaces.

Definition/Reference Records

These records contain the data for either a symbol definition or reference. It is composed of a filename field, a line number field, a type

field, a value field (omitted for references), and a symbol name field. These fields are defined as follows:

Filename Field

This field will be either an eight character filename or a hex X'22'. If a hex X'22', then the filename reference is the same as the previous record.

Line Number Field

This field contains the line number of the definition or reference statement in low-order high-order form.

Type Field

The type field contains an X'ØØ' for a reference, an X'Ø1' for a definition, or an X'Ø2' for a DEFL defined symbol.

Value Field

The value field contains the defined value of the symbol. This field is omitted for references (type field = Ø).

Name Field

The name field contains the symbol name. It is terminated with a carriage return (X'ØD'). If the symbol is the name of a macro, the first character of the name has the high-order bit set.

LINKAGE TO DEBUGGING (Model I/III or LDOS 6.x only)

In order to facilitate the debugging of user generated programs, a number of features have been built into EDAS. It provides the option of assembling source code directly to memory. It provides a command to transfer control to a user-supplied address (via the <B>ranch command).

A re-entry address to the Editor Assembler has been provided. If at any time during the debugging phase, you want to return to the Editor Assembler without reinitializing it (which would have deleted the entire text buffer), and are under the control of a debugging utility that does not utilize memory from X'54ØØ' (X'32ØØ' under LDOS 6.x) to the protected HIGH$, issue a jump command to X'58Ø3' (X'36Ø3' under LDOS 6.x). Alternately, you can provide a "JP 58Ø3H" (or JP 36Ø3H under LDOS 6.x) in your program as an exit and return to EDAS. A return to the Editor Assembler will be performed and the text buffer pointers will be maintained. If your program has maintained the integrity of the stack pointer, a RET instruction will return to the EDAS command prompt as the top of the stack contains the prompt address when an exit is made via the "B"ranch command.

EDAS disables the automatic entry to DEBUG on <BREAK> to avoid inadvertantly entering DEBUG by depressing <BREAK> to exit an <I>nsert or abort an assembly. In order to enter DEBUG directly from EDAS, perform a <B>ranch command to address X'3Ø'.

LC 1.1 Errata - 07/01/83

1.0 - cursor()

Correct the syntax of cursor() on page 4-34 and Appendix B-1 to read cursor(col,row);

2.0 #option errormsg

Reference the additional option, "errormsg" on pages 3-8 and 5-1/5-2. This option permits you to suppress the automatic display of operating system error messages generated on file I/O errors and generally displayed by the DOS @ERROR routine. Suppress the display by specifying "#option errormsg OFF". The default of errormsg is ON meaning system error messages will be displayed.

3.0 #option getnl

Reference the additional option, "getnl" on pages 3-8 and 5-1/5-2. This option refers to the handling of the newline character (\n) within the fgets() function. According to K&R, the newline should be included in the buffer returned by fgets(). Under LC 1.0, newline was stripped from the buffer. LC 1.1 has been brought into agreement with K&R on this point. If "#option getnl OFF" is specified, then fgets() will strip the newline as under LC 1.0. If you have programs compiled under 1.0 that require the newline be stripped, you can either reprogram your C source to coincide with the language as K&R specify, or you can specify the option as shown.

4.0 automatic variables

Automatic variable names may be re-used within a nested block. LC 1.0 did not support this although the LC manual did not reflect the limitation.